Improving Data Quality: Data Consistency, Deduplication, Currency and Accuracy

Wenyuan Yu

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2013

Abstract

Data quality is one of the key problems in data management. An unprecedented amount of data has been accumulated and has become a valuable asset of an organization. The value of the data relies greatly on its *quality*. However, data is often dirty in real life. It may be inconsistent, duplicated, stale, inaccurate or incomplete, which can reduce its usability and increase the cost of businesses. Consequently the need for improving data quality arises, which comprises of five central issues of improving data quality, namely, data consistency, data deduplication, data currency, data accuracy and information completeness. This thesis presents the results of our work on the first four issues with regards to data consistency, deduplication, currency and accuracy.

The first part of the thesis investigates incremental verifications of data consistencies in distributed data. Given a distributed database D, a set Σ of conditional functional dependencies (CFDs), the set V of violations of the CFDs in D, and updates ΔD to D, it is to find, with minimum data shipment, changes $\Delta\mathsf{V}$ to V in response to ΔD. Although the problems are intractable, we show that they are *bounded*: there exist algorithms to detect errors such that their computational cost and data shipment are both *linear* in the size of ΔD and $\Delta\mathsf{V}$, *independent of* the size of the database D. Such incremental algorithms are provided for both vertically and horizontally partitioned data, and we show that the algorithms are optimal.

The second part of the thesis studies the interaction between record matching and data repairing. Record matching, the main technique underlying data deduplication, aims to identify tuples that refer to the same real-world object, and repairing is to make a database consistent by fixing errors in the data using constraints. These are treated as separate processes in most data cleaning systems, based on heuristic solutions. However, our studies show that repairing can effectively help us identify matches, and vice versa. To capture the interaction, a uniform framework that seamlessly unifies repairing and matching operations is proposed to clean a database based on integrity constraints, matching rules and master data.

The third part of the thesis presents our study of finding *certain fixes* that are absolutely correct for data repairing. Data repairing methods based on integrity constraints are normally heuristic, and they may not find certain fixes. Worse still, they may even introduce new errors when attempting to repair the data, which may not work well when repairing *critical data* such as medical records, in which a seemingly minor error often has disastrous consequences. We propose a framework and an algorithm to

find certain fixes, based on master data, a class of *editing rules* and user interactions. A prototype system is also developed.

The fourth part of the thesis introduces inferring *data currency* and *consistency* for *conflict resolution*, where data currency aims to identify the current values of entities, and conflict resolution is to combine tuples that pertain to the same real-world entity into a single tuple and resolve conflicts, which is also an important issue for data deduplication. We show that data currency and consistency help each other in resolving conflicts. We study a number of associated fundamental problems, and develop an approach for conflict resolution by inferring data currency and consistency.

The last part of the thesis reports our study of *data accuracy* on the longstanding *relative accuracy problem* which is to determine, given tuples t_1 and t_2 that refer to the same entity e, whether $t_1[A]$ is more accurate than $t_2[A]$, *i.e.,* $t_1[A]$ is *closer to the true value* of the A attribute of e than $t_2[A]$. We introduce a class of accuracy rules and an inference system with a chase procedure to deduce relative accuracy, and the related fundamental problems are studied. We also propose a framework and algorithms for inferring accurate values with users' interaction.

Acknowledgements

First and foremost, I would like to express my deep gratitude to my supervisor, Professor Wenfei Fan, for all his support, guidance, patience, encouragement and inspiration. He led me into the area of databases study, and trained me the right way of doing research. Because of his hardworking, talent and preciseness, he is my esteemed role model in my scientific endeavour.

I am very grateful to my collaborators, including Yang Cao, Professor Floris Geerts, Professor Jianzhong Li, Professor Shuai Ma and Dr. Nan Tang. Thanks must also go to my colleagues, Dr. Xibei Jia, Dr. Yinghui Wu and Dr. Xin Wang, for their extended support. They have been very generous in sharing their rich and valuable knowledge and experience. It has been a happy time to work around people like them.

I thank the examiners of my thesis: Professor Peter Buneman and Professor Marcelo Arenas, for taking time to read my thesis and providing encouraging and constructive feedback. I am grateful for their thoughtful and detailed comments.

I thank all my fellow colleagues in the database group, for the helpful seminars, discussions, and all the fun memories I have had through the years.

Finally, I would like to thank my family: my parents for their unconditional support throughout my life; and my wife Cheng for her love, encouragement and understanding. Their support and encouragement was in the end what made this thesis possible.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Wenyuan Yu*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

We have entered an era of *Big Data*, with 2.5 quintillion (2.5×10^{18}) bytes of data being created everyday [IBM12]. Big Data is all about better analytics on a broader spectrum of data, and therefore represents an opportunity to create greater value for businesses [PCZ12]. Besides the challenges of dealing with the unprecedented *quantity of data*, it also poses the challenges which arise with *the quality of data* that, in order to get better analytics of data, the data is expected to be consistent, without duplication, up-to-date, accurate, and complete.

Unfortunately, the real-world data is often dirty: inconsistent, duplicated, stale, inaccurate or incomplete. Indeed, data error rates of approximately 1%–5% are typically found in enterprises, and for some companies it is above 30% [Red98]. In most data warehouse projects, it takes 30%-80% of the development time and budget [ST98], to improve the quality of the data instead of developing the systems. When it comes to incomplete information, it is estimated that "pieces of information perceived as being needed for clinical decisions were missing from 13.6% to 81% of the time" [MJYE05]. As for data currency, it is believed that "2% of records in a customer file become obsolete in one month" [Eck02]. That is to say, in a database of 500 000 customer records, 10 000 records may become stale per month, 120 000 records per year, and in two years about 50% of all the records may become obsolete [FG12]. Dirty data is also costly. It is known that dirty data costs US businesses 600 billion dollars annually [Eck02], and that erroneously priced data in retail databases costs US consumers $2.5 billion each year [Eng00]. These facts highlight the need for improving data quality.

1.1 Central Issues of Data Quality

There are five central issues of improving data quality that have been identified in [FG12], namely, data consistency, data deduplication, data currency, data accuracy and information completeness.

Data Consistency. It refers to the validity and integrity of data representing real-world entities [FG12]. It aims to find out inconsistencies in the data. Inconsistencies may exist in a single tuple, across multiple tuples in the same relation, or between tuples across multiple relations.

Data dependencies (*a.k.a.* integrity constraints [AHV95]) have been developed to ensure the consistency of the data, such as functional dependencies (FDs) [Cod72], conditional functional dependencies (CFDs) [BFG$^+$07] and inclusion dependencies (INDs) [CFP84]. Inconsistencies typically emerge as violations of data dependencies.

Data consistency involves two tasks: detection and repairing, where detection is to identify the violations of data dependencies, and repairing is to fix errors (inconsistencies).

Data repairing is regarded as one of the two central tasks for *data cleaning* [FLM$^+$11a], along with *record matching* (see below).

We study the the detection problem in the incremental and distributed setting in Chapter 2. We also study the data repairing problem in Chapter 3 and Chapter 4, where in Chapter 3, we focus on data monitoring (*i.e.,* repair data at the time of data entry) and guaranteeing correctness of fixes, while in Chapter 4, we investigate the interaction between the two central data cleaning tasks: data repairing and record matching.

Data Deduplication. It aims to identify a single tuple with true values to replace duplicate tuples that refer to the same real-world entity. It consists of record matching and conflict resolution.

(1) Record matching aims to identify tuples in one or more relations that refer to the same real-world entity. Along with data repairing, it is regarded as the other central task of data cleaning.

Record matching is a fundamental process for eliminating the duplicate records in a database. There has been a host of work on record matching (*e.g.,* [ARS09, BSIBD09, CGGM03, FJLM09, HS98, WBGM09]); see [EIV07, HSW09a] for surveys. Most data cleaning systems on the market support record matching, with some providing the functionality of data repairing at the same time.

(2) Conflict resolution aims to resolve conflicts by finding a single tuple with the true values to represent the entity.

Conflict resolution is challenging because it involves other central issues of data quality: the identified true values should be correct, consistent, up-to-date, and complete. It has been studied for decades, starting from [Day83]. It aims to combine data from different sources into a single representation (see [BN08, DN09] for surveys). In that context, inconsistencies are typically resolved by selecting the $\mathsf{max}, \mathsf{min}, \mathsf{avg}, \mathsf{any}$ value [BN08].

As mentioned earlier, in Chapter 4, we study the interaction between record matching and data repairing. We present our findings on conflict resolution by utilizing data consistency and data currency (see below) in Chapter 5.

Data Currency. It is also known as *timeliness*. It aims to identify the current values of entities represented by tuples in a database, and to answer queries with the current values [FG12].

The problem of data currency would be trivial when data values carried valid timestamps. In practice, however, timestamps are often unavailable or inaccurate [ZDI10]. Worse still, data values are often copied or imported from other sources [BESD$^+$09, DBEHS10, DBES09a, DBES09b], which may not share a uniform scheme of timestamps. These make it much more challenging to identify the "latest" values of entities from the data in a database.

Data currency has been studied in [FGW11], and a notion of currency orders and denial constraints has been used to infer the data currency. However, most cases identified in [FGW11] are intractable. This highlights the quest for a new class of simplified constraints and effective methods for identifying data currency.

We propose a new class of constraints and effective methods for inferring data currency. Also we use these constraints and methods with integrity constraints for conflict resolution. We report our findings in Chapter 5.

Data Accuracy. It refers to the closeness of values in a database to the true values of the entities that the data in the database represent [FG12]. The *relative accuracy problem* is to determine, given tuples t_1 and t_2 that refer to the same entity e, whether $t_1[A]$ is more accurate than $t_2[A]$, *i.e.,* $t_1[A]$ is *closer to the true value* of the A attribute of e than $t_2[A]$. This has been a longstanding issue for data quality, and is challenging when the true values of e are unknown. The need for studying this is evident in decision making [Gel10], information systems [Epp06] and data quality management [FLR94].

While data accuracy has long been advocated [BS06, FLM09, NJE$^+$09], most of the previous studies have focused on metrics for accuracy measurement; we are not aware of any formal treatment of relative accuracy in the absence of true values.

In Chapter 6, we present our study on inferring relative accuracy.

Information Completeness. It concerns whether our database has the complete information to answer our queries [FG12]. Given a database D and a query Q, we want to know whether Q can be completely answered by using only the data in D. If the information in D is incomplete, the answer to Q can hardly be accurate or even correct.

In practice the databases often do not have sufficient information to answer queries at hand. Both attribute values and tuples may be missing from our databases. Incomplete information introduces serious problems to enterprises, ranging from misleading analytical results to biased decisions, which leads to loss of credibility, customers and revenues[FG12]. The problems of information completeness has been studied in [FG10, RN11, SPPN12].

Information completeness is not studied in this thesis.

1.2 Thesis Organization

The reminder chapters of this thesis is organized as follows.

(2) Incremental Detection of Inconsistencies in Distributed Data. In Chapter 2, we first investigate incremental detection of errors in distributed data. Given a distributed database D, a set Σ of conditional functional dependencies (CFDs), the set V of violations of the CFDs in D, and updates ΔD to D, it is to find, with minimum data shipment, changes $\Delta\mathsf{V}$ to V in response to ΔD. The need for the study is evident since real-life data is often dirty, distributed and frequently updated. It is often prohibitively expensive to recompute the entire set of violations when D is updated. We show that the incremental detection problem is NP-complete for database D that is partitioned either vertically or horizontally, even when Σ and D are fixed. Nevertheless, we show that it is *bounded*: there exist algorithms to detect errors such that their computational cost and data shipment are both *linear* in the size of ΔD and $\Delta\mathsf{V}$, *independent of* the size of the database D. We provide such incremental algorithms for vertically partitioned data and horizontally partitioned data, and show that the algorithms are optimal. We further propose optimization techniques for the incremental algorithm over vertical partitions to reduce data shipment. We experimentally verify, using real-life data on

Amazon Elastic Compute Cloud (EC2), that our algorithms substantially outperform their batch counterparts.

The problems studied in Chapter 2 are related to the issue of data consistency (detection).

(3) Towards Certain Fixes with Editing Rules and Master Data. In Chapter 3, we propose a method for finding certain fixes, based on master data, a notion of *certain regions*, and a class of *editing rules*. A certain region is a set of attributes that are assured correct by the users. Given a certain region and master data, editing rules tell us what attributes to fix and how to update them. We show how the method can be used in data monitoring and enrichment. We also develop techniques for reasoning about editing rules, to decide whether they lead to a unique fix and whether they are able to fix all the attributes in a tuple, *relative to* master data and a certain region. Furthermore, we present a framework and an algorithm to find certain fixes, by interacting with the users to ensure that one of the certain regions is correct. We experimentally verify the effectiveness and scalability of the algorithm.

The problems studied in Chapter 3 are related to the issue of data consistency (repairing).

(4) Interaction Between Record Matching and Data Repairing. Chapter 4 studies a new problem, namely, the interaction between record matching and data repairing. We show that repairing can effectively help us identify matches, and vice versa. To capture the interaction, we propose a uniform framework that seamlessly unifies repairing and matching operations, to clean a database based on integrity constraints, matching rules and master data. We give a full treatment of fundamental problems associated with data cleaning via matching and repairing, including the static analyses of constraints and rules taken together, and the complexity, termination and determinism analyses of data cleaning. We show that these problems are hard, ranging from NP- or coNP-complete, to PSPACE-complete. Nevertheless, we propose efficient algorithms to clean data via both matching and repairing. The algorithms find *deterministic fixes* and *reliable fixes* based on confidence and entropy analysis, respectively, which are more accurate than possible fixes generated by heuristics. We experimentally verify that our techniques significantly improve the accuracy of record matching and data repairing taken as separate processes, using real-life data.

The problems studied in Chapter 4 are related to the issue of data consistency (repairing) and data deduplication (record matching).

(5) Inferring Data Currency and Consistency for Conflict Resolution. Chapter 5 introduces a new approach for conflict resolution: given a set of tuples pertaining to the same entity, it is to identify a single tuple in which each attribute has *the latest and consistent* value in the set. This problem is important in data integration, data cleaning and query answering. It is, however, challenging since in practice, reliable timestamps are often absent, among other things. We propose a model for conflict resolution, by specifying data currency in terms of partial currency orders and currency constraints, and by enforcing data consistency with constant conditional functional dependencies. We show that identifying data currency orders helps us repair inconsistent data, and vice versa. We investigate a number of fundamental problems associated with conflict resolution, and establish their complexity. In addition, we introduce a framework and develop algorithms for conflict resolution, by integrating data currency and consistency inferences into a single process, and by interacting with users. We experimentally verify the accuracy and efficiency of our methods using real-life and synthetic data.

The problems studied in Chapter 5 are related to the issue of data consistency, data currency and data deduplication (conflict resolution).

(6) Determining the Relative Accuracy of Attributes. Chapter 6 proposes a model for determining relative accuracy. (1) We introduce a class of accuracy rules and an inference system with a chase procedure, to deduce relative accuracy. (2) We identify and study several fundamental problems for relative accuracy. Given a set I_e of tuples pertaining to the same entity e and a set of accuracy rules, these problems are to decide whether the chase process terminates, is Church-Rosser, and leads to a unique *target tuple* t_e composed of the most accurate values from I_e for all the attributes of e. (3) We propose a framework for inferring accurate values with user interaction. (4) We provide algorithms underlying the framework, to find the unique target tuple t_e whenever possible; when there is no enough information to decide a complete t_e, we compute top-k candidate targets based on a preference model. (5) Using real-life and synthetic data, we experimentally verify the effectiveness and efficiency of our method.

The problems studied in Chapter 6 are related to the issue of data accuracy.

(7) Conclusion and Future Work. Chapter 7 concludes the thesis and the future works to be done.

1.3 Contributions

In this thesis, the studies on the first four central issues of data quality, namely, data consistency, data deduplication, data currency, data accuracy and the interactions between them are reported.

Incremental Detection of Inconsistencies in Distributed Data. We establish the complexity bounds and provide efficient algorithms for incrementally detecting the violations of CFDs in fragmented and distributed data, either vertically or horizontally (Chapter 2).

1. We formulate incremental detection as an optimization problem, and establish its complexity bounds (Section 2.3). We show that the problem (decision version) is NP-complete even when both D and CFDs are fixed, *i.e.,* when only the size $|\Delta D|$ of updates varies. Nevertheless, we show that the problem is *bounded* [RR96]: there exist algorithms for incremental detection such that their communication costs and computational costs are functions in the size of *the changes* in the input and output (*i.e.,* $|\Delta D|$ and $|\Delta \mathrm{V}|$), *independent of the size of database D*. This tells us that incremental detection can be carried out efficiently, since in practice, ΔD and $\Delta \mathrm{V}$ are typically small.
2. We develop an algorithm for incrementally detecting violations of CFDs for vertical partitions (Section 2.4). We show that the algorithm is *optimal* [RR96]: both its communication costs and computational costs are *linear* in $|\Delta D|$ and $|\Delta \mathrm{V}|$. Indeed, $|\Delta D|$ and $|\Delta \mathrm{V}|$ characterize the amount of work that is *absolutely necessary* to perform for incremental detection [RR96].
3. We develop optimization methods (Section 2.5) to further reduce data shipment for error detection in vertical partitions. The idea is to identify and maximally share indices among CFDs such that when multiple CFDs demand the shipment of the same tuples, only a single copy of the data is shipped. We show that the problem (the decision version) for building optimal indices is NP-complete, but provide an efficient heuristic algorithm.
4. We also provide an incremental detection algorithm for horizontal partitions (Section 2.6). We show that the algorithm is also *optimal*, as for its vertical counterpart.
5. Using TPCH for large scale data and DBLP for real-life data, we conduct experiments on Amazon EC2. We find that our incremental algorithms outperform their batch counterparts by *two orders of magnitude*, for fairly large updates (up

to 10GB for TPCH). Moreover, our methods scale well with both the size of data and the number of CFDs. We also find the optimization strategies effective.

Towards Certain Fixes with Editing Rules and Master Data. We propose a method for data monitoring and certain fixes, by capitalizing on editing rules, master data and user interactions (Chapter 3).

1. We present an interactive framework to find certain fixes (Section 3.5). A set of certain regions are first recommended to the users, derived from a set Σ of editing rules and master data D_m available, by using an algorithm of [FLM$^+$10]. For an input tuple t, the users may only ensure that $t[X]$ is correct, for a set X of attributes of t. If $t[X]$ matches any of the certain regions, the rules guarantee to find t a certain fix. Otherwise we deduce what other attributes Y of t are implied correct by $t[X]$ and the rules, and moreover, suggest a *minimal* set S of attributes such that as long as $t[S]$ is assured correct, $Y \cup S$ covers a certain region and hence, a certain fix to the entire t is warranted. The interactive process proceeds until the users are guided to reach a certain region.
2. We show that it is NP-complete to find a minimum suggestion S (decision version). Nonetheless, we develop an efficient heuristic algorithm to find a set of suggestions, and introduce effective optimization techniques. These yield a practical data entry solution to clean data.
3. We develop CERFIX, a data cleaning system that finds certain fixes for input tuples at the point of data entry. It differs from other systems that also ask for user feedback (*e.g.,* [CCC$^+$10]) in what feedback is requested and how the feedback is used. CERFIX consists of (a) a region finder to identify certain regions; (b) a data monitor to find certain fixes for input tuples, by guiding users to validate a minimal number of attributes via a javascript plug-in that can be embedded in any web forms; and (c) an auditing module to show what attributes are fixed and where the correct values come from.
4. We experimentally verify the effectiveness and scalability of the algorithm, using real-life hospital data and DBLP (Section 3.7). We find that the algorithm effectively provides suggestions, such that most input tuples are fixed with two or three rounds of interactions only. We also show that it scales well with the size of master data, and moreover, that the optimization techniques effectively reduce the latency during interactions.

Interaction Between Record Matching and Data Repairing. We approach this problem by *unifying* record matching and data repairing, and to provide a data cleaning solution that stresses accuracy (Chapter 4).

1. We investigate a new problem, stated as follows.

   Given a database D, master data D_m, and data quality rules consisting of CFDs Σ and matching rules Γ, the *data cleaning problem* is to find a repair D_r of D such that (a) D_r is *consistent* (*i.e.,* satisfying the CFDs Σ), (b) no more tuples in D_r can be *matched* to master tuples in D_m by rules of Γ, and (c) D_r minimally differs from the original data D.

   As opposed to record matching and data repairing, the data cleaning problem aims to fix errors in the data by unifying matching and repairing, and by leveraging master data. Here *master data* (*a.k.a.* reference data) is a single repository of high-quality data that provides various applications with a synchronized, consistent view of its core business entities [Los09]. It is being widely used in industry, supported by, *e.g.,* IBM, SAP, Microsoft and Oracle. To identify tuples from D and D_m, we use matching rules that are an extension of MDs [FJLM09] by supporting negative rules (*e.g.,* a male and female may not refer to the same person) [ARS09, WBGM09].

2. We propose a uniform framework for data cleaning. We treat both CFDs and MDs as *cleaning rules*, which tell us how to fix errors. This yields a rule-based logical framework, which allows us to seamlessly interleave repairing and matching operations. To assure the accuracy of fixes, we make use of (a) the *confidence* placed by the user in the accuracy of the data, (b) *entropy* measuring the certainty of data, by the self-information of the data itself [CT91, SV10], and (c) master data [Los09]. We distinguish three classes of fixes: (i) *deterministic* fixes for the unique solution to correct an error; (ii) *reliable* fixes for those derived using entropy; and (iii) *possible* fixes for those generated by heuristics. The former two are more accurate than possible fixes.

3. We investigate fundamental problems associated with data cleaning via both matching and repairing. We show the following. (a) When CFDs and matching rules are taken together, the classical decision problems for dependencies, namely, the consistency and implication analyses, are NP-complete and coNP-complete, respectively. These problems have the same complexity as their counterparts for CFDs [FGJK08], *i.e.,* adding matching rules does not incur extra complexity. (b) The data cleaning problem is NP-complete. Worse still, it is approximation-hard, *i.e.,* it is beyond reach in practice to find a polynomial-

time (PTIME) algorithm with a constant approximation ratio [WP05] unless P = NP. (c) It is more challenging to decide whether a data cleaning process terminates and whether it yields deterministic fixes: these problems are both PSPACE-complete.

4. In light of the inherent complexity, we propose a three-phase solution consisting of three algorithms. (a) One algorithm identifies *deterministic fixes* that are accurate, based on confidence analysis and master data. (b) When confidence is low or unavailable, we provide another algorithm to compute *reliable fixes* by employing information entropy, inferring evidence from data itself to improve accuracy. (c) To fix the remaining errors, we extend the heuristic based method [CFG$^+$07] to find a consistent repair of the dirty data. These methods are complementary to each other, and can be used either alone or together.
5. We experimentally evaluate the quality and scalability of our data cleaning methods with both matching and repairing, using real-life datasets (DBLP and hospital data from US Dept. of Health & Human Services). We find that our methods substantially outperform matching and repairing taken as separate processes in the accuracy of fixes, up to 15% and 30%, respectively. Moreover, deterministic fixes and reliable fixes are far more accurate than fixes generated by heuristic methods. Despite the high complexity of the cleaning problem, we also find that our algorithms scale reasonably well with the size of the data.

Inferring Data Currency and Consistency for Conflict Resolution. We study conflict resolution by inferring *both* data currency *and* data consistency (Chapter 5).

1. We propose a model for conflict resolution (Section 5.2). We specify data currency in terms of (a) *partial currency orders* denoting available (yet possibly incomplete) temporal information on the data, and (b) simple *currency constraints*, to express currency relationships derived from the semantics of the data. Data consistency is specified in terms of *constant CFDs* [FGJK08] on the latest values of the data. Given such a specification S_e on a set E of tuples pertaining to the same entity e, we aim to derive the true values of e from S_e.
2. We introduce a framework for conflict resolution (Section 5.3). One may find *some* true values of an entity from a specification of an entity, but *not all*. In light of this, our framework *automatically derives* as many true values as possible from a given specification S_e of an entity e, identifies attributes for which the true values of e are not derivable from S_e, and *interacts* with users to solicit

additional input for those attributes, so that all the true values of all the attributes of e can be derived from S_e and users' input.

3. We study problems fundamental to conflict resolution (Section 5.4). Given a specification S_e, we determine whether partial currency orders, currency constraints and CFDs in S_e have conflicts among themselves? Whether some other currency orders are implied by S_e? Whether true values of an entity can be derived from S_e? If not, what additional minimum currency information has to be provided so that the true values are derivable? We establish their complexity bounds, ranging from NP-complete and coNP-complete to Σ_2^p-complete. These results reveal the complexity *inherent to* conflict resolution.
4. We develop several practical algorithms (Section 5.5). We propose methods for finding (a) whether a specification S_e has conflicts, (b) what true values can be derived from S_e, and (c) a minimum set of attributes that require users' input to find their true values. All these problems are *intractable*; in particular, the last problem is Σ_2^p-complete. Nevertheless, we provide efficient heuristic algorithms, by integrating inferences of data consistency and currency into a single process.
5. We evaluate the accuracy and efficiency of our method using real-life and synthetic data (Section 5.6). We find that *unifying* currency and consistency *substantially improves* the accuracy of traditional methods, by 201% (F-measure), even with only a small number of constraints. It is also more effective than taking consistency and currency *separately*. Furthermore, our algorithms are efficient, and scale well with the number of tuples pertaining to an entity and with the number of constraints; for example, it takes an average of 7 seconds to resolve conflicts in sets of 8k-10k tuples representing an entity, with 1983 constraints.

Determining the Relative Accuracy of Attributes. We make a first attempt to give a formal treatment of relative accuracy, from theory to practice (Chapter 6).

1. We propose a model for determining relative accuracy (Section 6.2). We introduce (a) *accuracy rules* (ARs) defined in terms of partial orders; and (b) a chase-like procedure [AHV95] that, given a set I_e of tuples pertaining to the same entity e, a set Σ of ARs on I_e and (partial) master data I_m, infers relative accuracy and a target tuple by applying the ARs.
2. We identify fundamental problems for relative accuracy (Section 6.3). Given I_e, Σ and I_m, these problems are to decide (a) whether the chase process on I_e terminates by applying Σ and I_m? (b) Whether do all the chase sequences lead

to a unique target tuple t_e for e from I_e, *i.e.,* Church-Rosser [AHV95], no matter in what orders the rules are applied? (c) If t_e is incomplete, *i.e.,* some of its attributes have the null value, can we make t_e complete while observing the ARs in Σ? We show that the chase process always terminates, the Church-Rosser property can be decided in $O((|I_e|^2 + |I_m|)|\Sigma|)$ time, whereas the last problem is NP-complete.

3. We present a framework for deducing target tuples (Section 6.4). Given I_e, Σ and I_m, the framework checks whether the chase on I_e with Σ and I_m is Church-Rosser. If so, it *automatically deduces* as many accurate attribute values for t_e as possible. If t_e is incomplete, it computes top-k candidate targets based on a preference model. The users may check the candidate tuples, revise t_e, I_e and Σ, and invoke the process again until a satisfactory target tuple is found.
4. We provide effective algorithms underlying the framework (Sections 6.5 and 6.6). We give an algorithm for deciding whether the chase is Church-Rosser given I_e, Σ and I_m, and deducing accurate attributes for target tuples. We also develop three algorithms for finding top-k candidate target tuples, with the *early termination* property without inspecting all possible tuples. In particular, one of the algorithms does not require ranked lists as input, and is *instance optimal w.r.t.* the number of visits to the data [FLN03].
5. We experimentally verify the effectiveness and efficiency of our method, using real-life and synthetic data (Section 6.7). We find that our approach is effective: for the real-life data, accurate values are automatically deduced for at least 73% of the attributes without user interaction, in 10 milliseconds (ms); moreover, at most 3-4 rounds of user interaction are needed to find complete target tuples. Our algorithms scale well with the sizes of entity instances, master data and ARs. We also evaluate our method for truth discovery, vs. prior approaches [DBES09b, FGTY13]. We show that our model can accommodate trust in data sources [DBES09b] as well as data currency and consistency [FGTY13]. Even for truth discovery, our method performs as well as [DBES09b, FGTY13] in their settings, or even better.

1.4 List of Publications

During the course of the PhD study, as a co-author, I have published the following publications. And they contain the work reported in this thesis.

- [CFY13] Yang Cao, Wenfei Fan, and Wenyuan Yu
  Determining the Relative Accuracy of Attributes ACM SIGMOD International Conference on Management of Data, SIGMOD 2013 (Work for this paper is reported in Chapter 6)
- [FGTY13] Wenfei Fan, Floris Geerts, Nan Tang, and Wenyuan Yu
  Inferring Data Currency and Consistency for Conflict Resolution 29th IEEE International Conference on Data Engineering (ICDE), 2013
- [FLTY14] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu
  Incremental Detection of Inconsistencies in Distributed Data IEEE Transaction on Knowledge and Data Engineering (TKDE) (Special issue: Best Papers of ICDE 2012, to appear)
- [FLM$^+$12] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu
  Towards certain fixes with editing rules and master data The VLDB Journal 21(2): 213-238, 2012

- [FLTY12] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu
  Incremental Detection of Inconsistencies in Distributed Data 28th IEEE International Conference on Data Engineering (ICDE), 2012
- [FLM$^+$11b] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu
  CerFix: A System for Cleaning Data with Certain Fixes Demo, The 37th International Conference on Very Large Data Bases (VLDB), 2011
  (Work for this demo is reported in Section 3.6)
- [FLM$^+$11a] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu
  Interaction between Record Matching and Data Repairing ACM SIGMOD Conference on Management of Data (SIGMOD), 2011
- [FLM$^+$10] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu
  Towards certain fixes with editing rules and master data The 36th International Conference on Very Large Data Bases (VLDB), 2010

Remark. It is worth mentioning that the (partial) results of this thesis appear in the above publications: (1) results in Chapter 2 appear in [FLTY14] and [FLTY12]; (2) results in Chapter 3 appear in [FLM$^+$12]; (3) results in Chapter 4 appear in [FLM$^+$11a]; (4) results in Chapter 5 appear in [FGTY13]; and (5) results in Chapter 6 appear in [CFY13].

Chapter 2

Incremental Detection of Inconsistencies in Distributed Data

2.1 Introduction

In this chapter, we investigate incremental detection of errors in distributed data. It is related to the issue of data consistency.

As remarked earlier, detection of inconsistencies is one of the fundamental tasks of data consistency. Detecting inconsistencies in a centralized database is often simple. For example, two SQL queries suffice to detect inconsistencies of CFDs for centralized data [FGJK08]. However, data is increasingly common to be *partitioned* vertically (*e.g.,* [Sto05]) or horizontally (*e.g.,* [Kal08]), and *distributed* across different sites. This is highlighted by the recent interests in SaaS and Cloud computing, MapReduce [DG04, NPM$^+$10] and DBMS with columnar storage [Sto05]. In the distributed settings, however, it is much harder to detect errors in the data than in the centralized setting.

Example 2.1.1: Consider an employee relation D_0 shown in Fig. 2.2, which consists of tuples t_1–t_5 (ignore t_6 for the moment), and is specified by the following schema:

$\mathsf{EMP}(\mathsf{id}, \mathsf{name}, \mathsf{sex}, \mathsf{grade}, \mathsf{street}, \mathsf{city}, \mathsf{zip}, \mathsf{CC}, \mathsf{AC}, \mathsf{phn}, \mathsf{salary}, \mathsf{hd})$

Each EMP tuple specifies the id, name, sex, salary grade level, address (street, city, zip code), phone number (country code CC, area code AC, phone phn), salary and the date hired (hd). Here the employee id is a *key* of EMP.

To detect errors, a set of CFDs is defined on the EMP relation, as shown in Fig. 2.1. Here ϕ_1 asserts that for employees in the UK (*i.e.,* CC = 44), zip code uniquely determines street. CFD ϕ_2 assures that for any UK employee, if the area code is 131 then the city must be EDI.

Errors in D_0 emerge as violations of the CFDs, *i.e.,* those tuples in D_0 that violate at least one CFD in Σ_0, as shown in Fig. 2.1. For instance, t_1 and t_5 violate ϕ_1: they represent UK employees with the same zip, but have different street's. Moreover, t_1 *alone* violates ϕ_2: $t_1[\mathsf{CC}] = 44$ and $t_1[\mathsf{AC}] = 131$, but $t_1[\mathsf{city}] = \text{'NYC'} \neq \text{'EDI'}$. When D_0 is in a centralized database, the violations can be easily caught by using SQL-based techniques [FGJK08].

Now consider distributed settings. As depicted in Fig. 2.2, D_0 is partitioned either (1) vertically into three fragments D_{V_1}, D_{V_2} (grey columns) and D_{V_3}, all with attribute id; or (2) horizontally into D_{H_1} (t_1–t_2), D_{H_2} (t_3–t_4) and D_{H_3} (t_5), for employees with salary grade 'A' (junior level), 'B' and 'C' (senior), respectively. The fragments are distributed over different sites.

To find violations in both settings, it is necessary to *ship data from one site to another*. For instance, to find the violations of ϕ_1 in the vertical partitions, one has to send tuples with CC = 44 from the site of D_{V_3} to the site of D_{V_2}, or the other way around to ship attributes (street, zip); similarly for the horizontal partitions. □

CFDs	Violations
ϕ_1 : ([CC = 44, zip] → [street])	t_1, t_3, t_4, t_5
ϕ_2 : ([CC = 44, AC = 131] → [city = 'EDI'])	t_1

Figure 2.1: Example CFDs and their violations

Given a distributed database D, a set Σ of conditional functional dependencies (CFDs), the set V of violations of the CFDs in D, and updates ΔD to D, it is to find, with minimum data shipment, changes $\Delta\mathsf{V}$ to V in response to ΔD. The need for the study is evident since real-life data is often dirty, distributed and frequently updated. It is often prohibitively expensive to recompute the entire violations when D is updated.

It is NP-complete to find violations of CFDs, with minimum data shipment, in a distributed relation that is partitioned either horizontally or vertically [FGMM10]. A heuristic algorithm was developed in [FGMM10] to compute the violations of CFDs in *horizontally* partitioned data, which takes 80 seconds to find violations of one CFD in 8 fragments (*i.e.,* 8 sites) of 1.6 million tuples [*].

Distributed data is also often *dynamic*, *i.e.,* frequently updated [OV99].

[*] Tested on Linux PCs with 1.86GHz Intel Core 2 CPU and 2GB memory.

It is often prohibitively expensive to recompute the entire violations in a distributed database D when D is updated. This motivates us to study *incremental detection* of errors. In a nutshell, let V denote the violations of a set Σ of CFDs in D, ΔD be updates to D, and $D \oplus \Delta D$ denote the database updated by ΔD. In contrast to *batch algorithms* that compute violations of Σ in D starting from scratch, incremental detection is to find *changes* $\Delta\mathsf{V}$ to V, which aims to minimize unnecessary recomputation. Indeed, when ΔD is small, $\Delta\mathsf{V}$ is often small as well, though $\Delta\mathsf{V}$ may include tuples from ΔD and D. It is more efficient to compute $\Delta\mathsf{V}$ than the entire violations of Σ in $D \oplus \Delta D$.

Example 2.1.2: Consider ϕ_1 of Fig. 2.1, relation D_0 and its partitions given in Fig. 2.2, and the updates below.

(1) Insertions. Assume that t_6 is inserted into D_0, as shown in Fig. 2.2. Then the new violation $\Delta\mathsf{V}$ is $\{t_6\}$.

(a) *Batch computation.* In the vertical partitions, one needs to ship either tuples with the same $(\mathsf{zip}, \mathsf{street})$ as t_6 (in D_{V_2}) or 6 tuples with $\mathsf{CC} = 44$ (D_{V_3}), as shown in Example 2.1.1. In the horizontal partition, we have to compare all tuples with $\mathsf{CC} = 44$, which requires the shipment of 4 (partial) tuples.

(b) *Incremental computation.* Since t_5 is already a violation of ϕ_1 in V and (t_5, t_6) together violate ϕ_1, we can conclude that t_6 is the only new violation of ϕ_1, *i.e.,* $\Delta\mathsf{V} = \{t_6\}$ for ϕ_1. Indeed, for any tuple t, if (t, t_6) violate ϕ_1, then either (t, t_5) violate ϕ_1 or $t[\mathsf{CC}, \mathsf{zip}, \mathsf{street}] = t_5[\mathsf{CC}, \mathsf{zip}, \mathsf{street}]$. In both cases, t is already in V (*i.e.,* a violation). Hence to find $\Delta\mathsf{V}$ for ϕ_1, one needs to ship a single tuple id in the vertical partition (Section 2.4), and no data to be shipped in the horizontal case (Section 2.6).

(2) Deletions. Assume that t_4 is deleted after the insertion of t_6. One can verify that only t_4 has to be removed from the violations of ϕ_1, *i.e.,* $\Delta\mathsf{V} = \{t_4\}$ for ϕ_1.

(a) *Batch computation.* To find violations of ϕ_1 in $D_0 \oplus \Delta D$, one has to ship the same amount of data as in (1)(a).

(b) *Incremental computation.* In contrast, since t_3, t_4 are both in V and $t_3[\mathsf{street}, \mathsf{zip}] = t_4[\mathsf{street}, \mathsf{zip}]$, one can verify that only t_4 should be removed from V. Indeed, for any t, if (t, t_4) violate ϕ_1, so do (t, t_3). Since t_3 remains in V, so does t. Again, one needs to ship a single tuple id in vertical partitions, and no data in the horizontal case. □

It has been verified in a number of applications that incremental algorithms are more efficient than their batch counterparts when updates are small [RR93]. This example shows that this holds for distributed error detection.

		D_{V_1}				D_{V_2}(with id replica)			D_{V_3}(with id replica)					Updates
		id	name	sex	grade	street	city	zip	CC	AC	phn	salary	hd	
D_{H_1}	t_1 :	1	Mike	M	A	Mayfield	NYC	EH4 8LE	44	131	8693784	65k	01/10/2005	
	t_2 :	2	Sam	M	A	Preston	EDI	EH2 4HF	44	131	8765432	65k	01/05/2009	
D_{H_2}	t_3 :	3	Molina	F	B	Mayfield	EDI	EH4 8LE	44	131	3456789	80k	01/03/2010	
	t_4 :	4	Philip	M	B	Mayfield	EDI	EH4 8LE	44	131	2909209	85k	01/05/2010	delete
D_{H_3}	t_5 :	5	Adam	M	C	Crichton	EDI	EH4 8LE	44	131	7478626	120k	01/05/1995	
	t_6 :	6	George	M	C	Mayfield	EDI	EH4 8LE	44	131	9595858	120k	01/07/1993	insert

Figure 2.2: An EMP relation D_0

This work provides fundamental results and a practical solution for error detection in distributed data. We focus on CFDs because they carry constant patterns and are difficult to handle, and moreover, as shown in [FGJK08], they capture inconsistencies that traditional dependencies fail to catch. The techniques developed here, nonetheless, can be readily used to incrementally detect violations of other dependencies used in data cleaning, such as functional dependencies and denial constraints. We discuss related work below, and review error detection in distributed data in Section 2.2.

Related work. Methods for (incrementally) detecting CFD violations are studied in [FGJK08] for centralized data, based on SQL techniques. There has been work on constraint enforcement in distributed databases (*e.g.,* [ADNR07, GW93, Huy97]). As observed in [GW93, Huy97], constraint checking is hard in distributed settings, and hence, certain conditions are imposed there so that their constraints can be checked locally at individual site, without data shipment. As shown by the examples above, however, to find CFD violations it is often necessary to ship data. Detecting constraint violations has been studied in [ADNR07] for monitoring distributed systems, which differs substantially from this work in that their constraints are defined on *system states* and cannot express CFDs. In contrast, CFDs are to detect errors in *data*, which is typically much larger than system states. Closer to this work is [FGMM10], which studies CFD violation detection in horizontal partitions, but considers neither incremental detection nor algorithms for detecting errors in vertical partitions.

Incremental algorithms have proved useful in a variety of areas (see [RR93] for a survey). In particular, incremental view maintenance has been extensively studied [GM99], notably for distributed data [BmLT86, GMS93, Rou91, BDMW98]. Various auxiliary structures have been proposed to reduce data shipment, *e.g.,* counters [BmLT86, GMS93], pointer [Rou91] and tags in base relations [BDMW98]. While these could be incorporated into our solution, they do not yield bounded/optimal incremental detection algorithms.

There has also been a host of work on query processing [Kos00] and multi-query optimization [KNCV08] for distributed data. The former typically aims to generate distributed query plans, to reduce data shipment or response time (see [Kos00] for a survey). Optimization strategies, *e.g.,* semiJoins [BC81], bloomJoins [ML86], and recently [LDK09, DT07, MN08, WBTD08], have proved useful in main-memory distributed databases (*e.g.,* MonetDB [FGKT10] and H-Store [Kal08]), and in cloud computing and MapReduce [DG04, NPM$^+$10]. Our algorithms leverage the techniques of [KNCV08] to reduce data shipment when validating multiple CFDs, in particular.

2.2 Error Detection in Distributed Data

In this section we review CFDs [FGJK08], data fragmentation [OV99] and error detection in distributed data [FGMM10].

2.2.1 Conditional Functional Dependencies

A CFD ϕ on relation R is a pair $(X \rightarrow Y, t_p)$, where (1) $X \rightarrow Y$ is a standard functional dependency (FD) on R; and (2) t_p is the *pattern tuple* of ϕ with attributes in X and Y, where for each attribute A in $X \cup Y$, $t_p[A]$ is either a constant in the domain $\mathsf{dom}(A)$ of A, or an unnamed variable '_' that draws values from $\mathsf{dom}(A)$ [OV99].

Example 2.2.1: The CFDs in Fig. 2.1 can be expressed as:

$$
\begin{array}{lll}
\phi_1: & ([\mathsf{CC}, \mathsf{zip}] \rightarrow [\mathsf{street}], & t_{p_1} = (44, \_, \_)) \\
\phi_2: & ([\mathsf{CC}, \mathsf{AC}] \rightarrow [\mathsf{city}], & t_{p_2} = (44, 131, \mathrm{EDI}))
\end{array}
$$

Note that FDs are a special case of CFDs in which the pattern tuple consists of '_' only. □

To give the semantics of CFDs, we use an operator $\asymp$ defined on constants and '_': $v_1 \asymp v_2$ if either $v_1 = v_2$, or one of v_1, v_2 is '_'. The operator extends to tuples, *e.g.,* $(131, \mathrm{EDI}) \asymp (\_, \mathrm{EDI})$ but $(131, \mathrm{EDI}) \not\asymp (\_, \mathrm{NYC})$.

An instance D of R *satisfies* a CFD ϕ, denoted by $D \models \phi$, iff for *all* tuples t and t' in D, if $t[X] = t'[X] \asymp t_p[X]$, then $t[Y] = t'[Y] \asymp t_p[Y]$. Intuitively, ϕ is defined on those tuples t in D such that $t[X]$ matches the pattern $t_p[X]$, and moreover, it enforces the pattern $t_p[Y]$ on $t[Y]$.

Example 2.2.2: Consider D_0 in Fig. 2.2 and the CFDs in Fig. 2.1. Then D_0 does not satisfy ϕ_1, since $t_1[\mathsf{CC}, \mathsf{zip}]$=$t_5[\mathsf{CC}, \mathsf{zip}] \asymp (44, \_)$ but $t_1[\mathsf{street}] \neq t_5[\mathsf{street}]$, violating ϕ_1. □

A set of CFDs of the form $(X \rightarrow Y, t_{p_i})$ $(i \in [1, n])$ can be converted to an equivalent form $(X \rightarrow Y, T_p)$, where T_p is a pattern tableau that contains n tuples $t_{p_1}, \cdots, t_{p_n}$ [FGJK08]. This is what we used in our implementation.

We call $(X \rightarrow B, t_p)$ a *constant* CFD if $t_p[B]$ is a constant, and a *variable* CFD if $t_p[B]$ is '_'. For instance, ϕ_2 in Fig. 2.1 is a constant CFD, while ϕ_1 is a variable CFD.

2.2.2 Data Fragmentation

We consider relations D of schema R that are partitioned into fragments, either vertically or horizontally.

Vertical partitions. In some applications (*e.g.,* [Sto05]) one wants to partition D into $(D_1, \ldots, D_n)$ [OV99] such that

$$D_i = \pi_{X_i}(D), \quad D = \bowtie_{i\in[1,n]} D_i,$$

where X_i is a set of attributes of R on which D is projected, including a *key* attribute of R. Relation D can be reconstructed by join operations on the *key* attribute.

Each vertical fragment D_i has its own schema R_i with attributes X_i. The set of attributes of R is $\bigcup_{i\in[1,n]} X_i$.

As shown in Fig. 2.2, D_0 can be partitioned vertically into D_{V_1}, D_{V_2} and D_{V_3}, where the schema of D_{V_1} is R_1(id, name, sex and grade); similarly for D_{V_2} and D_{V_3}.

Horizontal partitions. Relation D may also be partitioned (fragmented) into (D_1, ..., D_n) [OV99, Kal08] such that

$$D_i = \sigma_{F_i}(D), \quad D = \bigcup_{i\in[1,n]} D_i,$$

where F_i is a Boolean predicate and selection $\sigma_{F_i}(D)$ identifies fragment D_i. These fragments are disjoint, *i.e.,* no tuple t appears in distinct fragments D_i and D_j $(i \neq j)$. They have the same schema R. The original relation D can be reconstructed by the union of these fragments.

For example, D_0 is horizontally partitioned into D_{H_1}, D_{H_2} and D_{H_3} in Fig. 2.2, with the selection predicate as grade = ‘A’, grade = ‘B’ and grade = ‘C’, respectively.

2.2.3 Detecting CFD Violations in Distributed Data

When CFDs are used as data quality rules, errors in the data are captured as violations of CFDs [FGJK08, FGMM10].

Violations. For a CFD $\phi = (X \to Y, t_p)$ and an instance D of R, we use $\mathsf{V}(\phi, D)$ to denote the set of all tuples in D that violate ϕ, called the *violations of* ϕ *in D*. Here a tuple $t \in \mathsf{V}(\phi, D)$ iff there exists $t' \in D$ such that $t[X] = t'[X] \asymp t_p[X]$ but either $t[Y] \neq t'[Y]$ or $t[Y] = t'[Y] \not\asymp t_p[Y]$. For a set Σ of CFDs, we define $\mathsf{V}(\Sigma, D) = \bigcup_{\phi\in\Sigma} \mathsf{V}(\phi, D)$.

For instance, Fig. 2.1 lists violations of ϕ_1 and ϕ_2 in D_0.

When D is a centralized database, two SQL queries suffice to find $\mathsf{V}(\Sigma, D)$, no matter how many CFDs are in Σ. The SQL queries can be automatically generated [FGJK08].

Error detection in distributed data. Now consider a relation D that is partitioned into fragments $(D_1, \ldots, D_n)$, either vertically or horizontally. Assume *w.l.o.g.* that D_i's are distributed across distinct sites, *i.e.,* D_i resides at site S_i for $i \in [1, n]$, and S_i and S_j are distinct if $i \neq j$.

It becomes nontrivial to find $\mathsf{V}(\Sigma, D)$ when D is fragmented and distributed. As shown in Example 2.1.1, to detect the violations in distributed D_0, it is necessary to ship data from one site to another. Hence a natural question concerns how to find $\mathsf{V}(\Sigma, D)$ with minimum amount of data shipment. That is, we want to reduce communication cost and network traffic.

To characterize the communication cost, we use $M(i, j)$ to denote the set of tuples shipped from S_i to S_j, and M the total data shipment, *i.e.,* $\bigcup_{i,j \in [1,n], i \neq j} M(i, j)$.

For each $j \in [1, n]$, we use $D_j(M)$ to denote fragment D_j augmented by data shipped in M, *i.e.,* $D_j(M)$ includes data in D_j and all the tuples in M that are shipped to site S_j. More specifically, for vertical partitions,

$$D_j(M) = D_j \bowtie_{i \in [1,n] \wedge M(i,j) \neq \emptyset} M(i, j);$$

while for horizontal partitions,

$$D_j(M) = D_j \cup \bigcup_{i \in [1,n] \wedge M(i,j) \neq \emptyset} M(i, j).$$

We say that a CFD ϕ can be *checked locally after data shipments M* if $\mathsf{V}(\phi, D) = \bigcup_{i \in [1,n]} \mathsf{V}(\phi, D_i(M))$. As a special case, we say that ϕ can be *checked locally* if $\mathsf{V}(\phi, D) = \bigcup_{i \in [1,n]} \mathsf{V}(\phi, D_i)$, *i.e.,* all violations of ϕ in D can be found at individual site without data shipment (*i.e.,* $M = \emptyset$).

A set Σ of CFDs *can be checked locally after M* if each ϕ in Σ can be checked locally after M.

The *distributed CFD detection problem with minimum communication cost* is to determine, given a positive number K, a set Σ of CFDs and a partitioned and distributed relation D, whether there exists a set M of data shipments such that (1) Σ can be checked locally after M, and (2) the size $|M|$ of M is no larger than K, *i.e.,* $|M| \leq K$.

In contrast to the error detection problem in centralized data, it is beyond reach in practice to find an efficient algorithm to detect errors in distributed data with minimum network traffic [FGMM10].

Theorem 2.2.1:*[FGMM10] The distributed CFD detection problem with minimum communication cost is NP-complete, when data is either vertically or horizontally partitioned.* □

In light of the intractability, a heuristic algorithms was developed in [FGMM10] to compute $\mathsf{V}(\Sigma, D)$ when D is horizontally partitioned. We are not aware of any algorithm for detecting CFD violations for data that is vertically partitioned.

2.3 Incremental Detection: Complexity

We formulate the incremental detection problem and study its complexity. We start with notations for updates.

Updates. We consider a *batch update* ΔD to a database D, which is a list of tuple insertions and deletions. A modification is treated as an insertion after a deletion. We use ΔD^+ to denote the sub-list of all tuple insertions in ΔD, and ΔD^- the sub-list of deletions in ΔD. We use $D \oplus \Delta D$ to denote the updated database of D with ΔD.

In a vertical partition $D = (D_1, \ldots, D_n)$ (see Section 2.2), we write $\Delta D_i = \pi_{X_i}(\Delta D)$ for updates in ΔD to fragment D_i. For a horizontal partition, we denote the updates to D_i as $\Delta D_i = \sigma_{F_i}(\Delta D)$; similarly for ΔD_i^+ and ΔD_i^-.

Problem statement. Given D, ΔD and a set Σ of CFDs, we want to find $\mathsf{V}(\Sigma, D \oplus \Delta D)$, *i.e.,* all violations of CFDs of Σ in the updated database $D \oplus \Delta D$.

As remarked earlier, we want to minimize unnecessary recomputation by *incrementally* computing $\mathsf{V}(\Sigma, D \oplus \Delta D)$. More specifically, suppose that the old output $\mathsf{V}(\Sigma, D)$ is also provided. *Incremental detection* is to find the *changes* $\Delta\mathsf{V}$ to $\mathsf{V}(\Sigma, D)$ such that $\mathsf{V}(\Sigma, D \oplus \Delta D) = \mathsf{V}(\Sigma, D) \oplus \Delta\mathsf{V}$. We refer to this as the *incremental detection problem*.

In practice, when ΔD is small, $\Delta\mathsf{V}$ is often small as well. Hence it is more efficient to find $\Delta\mathsf{V}$ rather than *batch detection* that recomputes $\mathsf{V}(\Sigma, D \oplus \Delta D)$ starting from scratch. That is, we maximally reuse the old output $\mathsf{V}(\Sigma, D)$ when computing the new output $\mathsf{V}(\Sigma, D \oplus \Delta D)$.

We use $\Delta\mathsf{V}^+$ to denote $\mathsf{V}(\Sigma, D \oplus \Delta D) \setminus \mathsf{V}(\Sigma, D)$, *i.e.,* violations added, and $\Delta\mathsf{V}^-$ for $\mathsf{V}(\Sigma, D) \setminus \mathsf{V}(\Sigma, D \oplus \Delta D)$, *i.e.,* violations removed. Then $\Delta\mathsf{V} = \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}^-$. Observe that ΔD^+ only incurs $\Delta\mathsf{V}^+$, and ΔD^- only leads to $\Delta\mathsf{V}^-$.

When D is partitioned into $(D_1, \ldots, D_n)$ and distributed, we say that $\Delta\mathsf{V}$ can be *computed locally* after data shipments M of tuples from $D \oplus \Delta D$ if $\Delta\mathsf{V} = \bigcup_{i \in [1,n]} \Delta\mathsf{V}_i(M)$, where $\Delta\mathsf{V}_i(M)$ denotes the differences between $V(\Sigma, D_i(M) \oplus \Delta D_i)$ and $V(\Sigma, D_i)$ at site

S_i.

The *incremental* distributed CFD detection problem with minimum communication cost is to find, given D, Σ, $\Delta D, \mathsf{V}(\Sigma, D)$ as input, $\Delta\mathsf{V}$ with *minimum* data shipments M such that $\Delta\mathsf{V}$ is locally computable after M.

Its decision problem is to determine, given D, Σ, $\Delta D, \mathsf{V}(\Sigma, D)$ and a positive number K, whether there exists a set M of data shipments such that (1) $\Delta\mathsf{V}$ can be computed locally after M, and (2) $|M| \leq K$. We refer to the problem as IMVD for vertically partitioned data, and as IMHD for horizontally partitioned data.

In practice, the set Σ of CFDs is typically predefined and is rarely changed, although D is frequently updated. Thus in the sequel we consider fixed Σ.

Intractability results. Unfortunately, incremental detection is no easier than its batch counterpart (Theorem 2.2.1). Below we shall first study the case for vertical partitions, then analyze its horizontal counterpart.

Theorem 2.3.1: *The incremental distributed CFD detection problem with minimum data shipment is NP-complete for vertical partitions (*IMVD*). It remains NP-hard for fixed CFDs when (a) update consists of insertions only, for a fixed database with fixed partitions, or (b) update consists of deletions only.* □

Proof. [*] Upper bound. To show that IMVD is in NP, we provide an NP algorithm for incremental detection of violations in vertical partitions. It works as follows: first guess a set M of data shipments such that $|M| \leq K$, and then inspect whether $\Delta\mathsf{V} = \bigcup_{i\in[1,n]} \Delta\mathsf{V}_i(M)$. The checking can be done in PTIME.

Lower bound. We show that IMVD is NP-hard even when (1) ΔD consists of insertions only, or (2) ΔD consists of deletions only. We use fixed CFDs in both cases.

(1) When ΔD consists of insertions only. We verify the NP-hardness of IMVD by reduction from the minimum vertical detection problem (MVD). Given a set Σ of CFDs, a vertically partitioned database D and a positive number K, MVD is to decide whether there exists a set M of data shipments such that Σ can be checked locally after M, and $|M| \leq K$. It is known that MVD is NP-complete for a *fixed set* Σ defined on a fixed schema [FGMM10].

Given an instance (Σ, D, K) of MVD, we construct an instance $(\Sigma, D', \mathsf{V}(\Sigma, D), \Delta D^+, K)$ of IMVD by letting $D' = \emptyset$, $\Delta D^+ = D'$ and $\mathsf{V}(\Sigma, D) = \emptyset$. One can verify that there is M such that $|M| \leq K$ and Σ can be checked locally after M iff

[*] This proof is a joint work with Wenfei Fan.

there exists a set M' of data shipments such that $|M'| \leq K$ and $\Delta\mathsf{V}$ can be computed locally after M'. Note that $D' = \emptyset$ is independent of input (Σ, D, K). In other words, IMVD is NP-hard when the CFDs, the database and its partition are all fixed.

(2) When ΔD consists of deletions only. We show the NP-hardness of IMVD also by reduction from MVD. Given an instance (Σ, D, K) of MVD, we define an IMVD instance as follows. Assume that Σ is defined on schema R.

(a) We define a new schema $R' = R \cup \{B_1, B_2\}$, where B_1 and B_2 are distinct attributes not appearing in R.

(b) We define the set of $\Sigma' = \Sigma \cup \{\varphi\}$, where φ is an FD $B_1 \to B_2$. Assume *w.l.o.g.* that there exist two distinct values v_1 and v_2 in the domains of B_1 and B_2.

(c) We define D' such that for each $t_i \in D$, D' includes two tuples t_{ai} and t_{bi}, where $t_{ai}[R] = t_{bi}[R] = t[R]$, $t_{ai}[B_1 B_2] = (v_1, v_1)$, and $t_{bi}[B_1 B_2] = (v_1, v_2)$. That is, if D consists of n tuples. D' consists of $2 * n$ tuples. The relations D and D' have the same partitions for all the attributes in R. In addition, a new fragment of D' is added, consisting of new attributes B_1, B_2 and the key attribute key of D. Obviously, $\mathsf{V}(\Sigma, D') = D'$, since every tuple of D' violates φ with another tuple in D'.

(d) We define the set ΔD^- of *deletions* to be $\{t_{bi} \mid i \in [1, n]\}$, *i.e.,* it is to remove all tuples t_{bi}.

To see that these make a reduction, observe the following. Before D is updated by ΔD^-, $\mathsf{V}(\Sigma, D') = D'$. After D is updated, $\mathsf{V}(\Sigma, D' \oplus \Delta D^-)$ = $\mathsf{V}(\Sigma, D)$. From this it follows that a solution (a set of data shipments) to (Σ, D, K) iff it is a solution to $(\Sigma, D', \mathsf{V}(\Sigma, D), \Delta D^-, K)$. Moreover, since MVD is NP-complete when Σ and fragmentation are fixed, so is IMVD when ΔD consists of deletions only, since the newly added φ and the refined fragmentation are also independent of the input. □

We next analyze the case for horizontal partitions.

Theorem 2.3.2: *The incremental distributed CFD detection problem with minimum data shipment is NP-complete for horizontally partitioned data (*IMHD*). It remains NP-hard for fixed CFDs and for (a) insertions only, with a fixed database with fixed partitions, or (b) for deletions only.* □

Proof. [†] Upper bound. We show that IMHD is in NP by providing an NP algorithm for IMHD. It works as follows: first guess a set M of data shipments such that $|M| \leq K$,

[†] This proof is a joint work with Wenfei Fan.

and then inspect whether $\Delta\mathsf{V} = \bigcup_{i\in[1,n]} \Delta\mathsf{V}_i(M)$. The latter can be done in PTIME.

Lower bound. We show that IMHD is NP-hard for fixed CFDs even when (1) ΔD consists of insertions only with a fixed D, or (2) ΔD consists of deletions only.

(1) When ΔD consists of insertions only. We show that IMHD is NP-hard by reduction from the minimum set cover problem (MSC). Given a finite set X of elements, a collection $\mathcal{C}$ of subsets of X and a positive number K, MSC is to decide whether there exists a cover for X of size K or less, *i.e.,* a subset $\mathcal{C}' \subseteq \mathcal{C}$ such that $|\mathcal{C}'| \le K$ and every element of X belongs to at least one member of $\mathcal{C}'$. It is known that MSC is NP-complete even when each subset in $\mathcal{C}$ has three elements (cf. [GJ79]).

Given an instance $(X, \mathcal{C}, K)$ of MSC, we construct an instance (Σ, D, $\mathsf{V}(\Sigma, D)$, ΔD^+, K') of IMHD such that the IMHD problem has a solution iff the MSC problem has a solution. Assume *w.l.o.g.* that $X = \{x_j \mid j \in [1,m]\}$, $\mathcal{C} = \{C_i \mid i \in [1,n]\}$, each C_i consists of three elements of X, and that $X = \bigcup_{i\in[1,n]} C_i$ (*i.e.,* there exists a cover).

(a) We define schema $R = (A_1, A_2, A_3, B, N, L)$. Intuitively, A_1, A_2, A_3 are to encode the three elements in a subset C_i of $\mathcal{C}$, B for type (*i.e.,* a subset or an element), N is a partition key, and L is a tuple id within the fragment.

(b) The set Σ consists of three fixed FDs: $A_i \to B$, $i \in [1,3]$.

(c) We construct an instance D of R that is horizontally partitioned into 2 fragments D_u and D_v, residing at sites S_u and S_v, respectively. Assume an arbitrary topological order $\prec$ on the elements of X, and four fixed distinct values b_1, b_2, u and v. Tuples in D are partitioned into D_u and D_v with the selection predicate as $N = u$ and $N = v$, respectively. Initially, D is empty, and hence, both D_u and D_v are empty. Thus so are $\mathsf{V}(\Sigma, D_u)$ and $\mathsf{V}(\Sigma, D_v)$.

(d) We define *insertions* ΔD^+ as follows.

- ∘ ΔD_u^+ consists of $(n+m)$ tuples. For each $i \in [1,n]$, there exists a tuple t_{c_i} in ΔD_u^+ such that $t_{c_i} = (a_1, a_2, a_3, b_1, u, i)$, where a_1, a_2, a_3 are the elements in $C_i \in \mathcal{C}$, such sorted that $a_1 \prec a_2 \prec a_3$. For each i in $[1,m]$, there exists a tuple t_{x_i} in ΔD_u^+, such that $t_{x_i} = (x_i, x_i, x_i, b_2, u, i+n)$. Intuitively, each t_{c_i} encodes a subset C_i, and each t_{x_i} encodes an element of X.
- ∘ ΔD_v^+ consists of $m*(n+1)$ tuples. For each $i \in [1,m]$, there exist $(n+1)$ tuples t_{x_i1}, t_{x_i2} …, $t_{x_i(n+1)}$ in ΔD_v^+, such that $t_{x_ij} = (x_i, x_i, x_i, b_2, v, (i-1)*(n+1)+j)$, for $j \in [1, n+1]$. Intuitively, for each $i \in [1,m]$, there exist $(n+1)$ tuples that

encode x_i.

Assume *w.l.o.g.* that tuples in ΔD^+ have the same size l.

(e) We define K' to be $K * l$.

Observe that schema R, database D and CFDs Σ are all *fixed*, *i.e.,* they are independent of the MSC instance.

Intuitively, for all tuples $t \in \Delta D^+$, if $t[B] = b_1$, then t encodes a subset $C_i \in \mathcal{C}$; and if $t[B] = b_2$, then t encodes an element x_i in X. In addition, t_1 and t_2 in ΔD^+ violate a CFD of Σ if one of them is a tuple encoding a subset C_i, the other encodes an element x_i, and $x_i \in C_i$. All the tuples in ΔD_u^+ and ΔD_v^+ violate some CFDs of Σ. Note that only violations incurred by tuples t_{x_i} and t_{c_j} in ΔD_u^+ can be detected locally, without requiring data shipment. Tuples in ΔD_v^+ do not cause local violations; but for each tuple $t_{x_i j}$ there exists a tuple t_{c_k} in ΔD_u^+ such that $t_{x_i j}$ and t_{c_k} violate a CFD, where x_i is an element of C_k, $i \in [1,m], j \in [1,n+1]$, and $k \in [1,n]$. Intuitively, to detect violations in ΔD_v^+ locally, a *"cover"* $C' \subseteq \mathcal{C}$ of X must be shipped from site S_u to S_v.

We now show that $(\Sigma, D, \mathsf{V}(\Sigma, D), \Delta D^+, K')$ is indeed a reduction from MSC to IMHD. First, assume that the MSC instance has a cover C' of size no larger than K. We define a set M of tuple shipments $M = \{t_{c_i} \mid C_i \in C'\}$. We ship M from site S_u to S_v. Note that the size of M is no larger than K'. Since C' is a cover, at site S_v, all tuples $t \in D_v(M) \oplus \Delta D_v^+$ can be detected as violations locally. Hence, $\Delta\mathsf{V}_u(M) \cup \Delta\mathsf{V}_v(M) = \Delta\mathsf{V}_u \cup \Delta\mathsf{V}_v(M) = \Delta D_u^+ \cup \Delta D_v^+ \cup M = \Delta D_u^+ \cup \Delta D_v^+ = \Delta\mathsf{V}$.

Conversely, assume that there exists a set M of tuple shipments such that $|M| \leq K' = K * l$, and after M, $\Delta\mathsf{V}$ can be computed locally. (a) If $K' = n * l$, then the set $\mathcal{C}$ consisting of all subsets is a cover and $|\mathcal{C}| \leq n \leq K$. (b) When $K' < n * l$, let $M = M_{u \to v} \cup M_{v \to u}$, where $M_{u \to v}$ (resp. $M_{v \to u}$) denotes the part of M shipped from S_u (resp. S_v) to S_v (resp. S_u). Since $|M_{v \to u}| \leq |M| \leq K'$, there are no more than n tuples in $M_{v \to u}$. Thus for any element $x_i \in X$, there exists at least one tuple $t_{x_i j} \in \Delta D_v^+ \setminus M_{v \to u}$. Since each $t_{x_i j}$ is detected as a local violation, each x_i has to be covered by tuple t_{c_k} in $M_{u \to v}$, which encodes a subset C_k. Let $C' = \{C_k \mid t_{c_k} \in M_{u \to v}\}$. Then C' is indeed a cover of X, and $|C'| \leq K$.

(2) When ΔD consists of deletions only. We show that IMHD is NP-hard also by reduction from MSC.

Given an instance (X, C, K) of MSC, we construct an instance $(\Sigma, D', \mathsf{V}(\Sigma, D'), \Delta D^-, K')$ such that the IMHD problem has a solution iff MSC has a solution.

We use the same R, Σ and K' as defined in (1) above. An instance D' is also partitioned into D'_u and D'_v with the same predicates given in (1). More specifically,

- ○ $D'_u = \Delta D^+_u$, consisting of $(n+m)$ tuples given in (1);
- ○ D'_v consists of $(m*(n+1)+n)$ tuples, in which $m*(n+1)$ tuples are from ΔD^+_v given in (1). The other n tuples are given as follows. For each $i \in [1,n]$, D'_v includes a tuple $t'_{c_i} = (a_1, a_2, a_3, b_1, v, m*(n+1)+i)$, where a_1, a_2, a_3 are the elements in $C_i \in \mathcal{C}$, such sorted that $a_1 \prec a_2 \prec a_3$ for some order $\prec$.

We define *deletions* ΔD^- to be $\{t'_{c_i} \mid i \in [1,n]\}$, *i.e.,* it is to remove all those tuples t'_{c_i} from D'_v. Here $\mathsf{V}(\Sigma, D') = D'$, *i.e.,* every tuple in D' is a violation of some CFD in Σ.

Note that schema R and CFDs Σ are both fixed, *i.e.,* they are independent of the MSC instance.

Observe that before D' is updated by ΔD^-, all the violations can be detected locally in D'_u and D'_v. After D' is updated, $D' \oplus \Delta D^-$ became the relation D given in (1) above, and $\mathsf{V}(\Sigma, D' \oplus \Delta D^-) = \mathsf{V}(\Sigma, D)$. Hence along the same lines as the proof for (1), one can verify that $(\Sigma, D', \mathsf{V}(\Sigma, D'), \Delta D^-, K')$ is a reduction from MSC. □

From the proofs of Theorem 2.3.1 and 2.3.2, it follows:

Corollary 2.3.3: *The incremental distributed CFD detection problems* IMVD *and* IMHD *with minimum data shipment remains NP-complete even for fixed FDs only.* □

The boundedness result. Not all is lost. As observed in [RR96], the cost of an *incremental algorithm* should be analyzed in terms of the size of the *changes* in both input and output, denoted as $|\Delta\mathsf{C}|$, rather than the size of the entire input. Indeed, $|\Delta\mathsf{C}|$ characterizes the updating costs *inherent to* the incremental problem itself.

An incremental problem is said to be *bounded* if its cost can be expressed as a function of $|\Delta\mathsf{C}|$. An incremental algorithm is *optimal* if its cost is in $O(|\Delta\mathsf{C}|)$; *i.e.,* it only does the amount of work that is *necessary* to be performed by any incremental algorithm for the problem. In other words, it is the best one can hope for.

For incremental violation detection, $|\Delta\mathsf{C}| = |\Delta D| + |\Delta\mathsf{V}|$. It is *bounded* if its communication and computational costs are both functions of $|\Delta\mathsf{C}|$, *independent of* $|D|$.

Although the distributed incremental detection problem is NP-complete *w.r.t.* minimum data shipment (Theorems 2.3.1 and 2.3.2), the good news is that it is bounded *w.r.t.* the changes in both input and output.

Theorem 2.3.4: *The incremental distributed CFD detection problem is bounded for* *data partitioned vertically or horizontally. There are optimal incremental detection algorithms with communication and computational costs in* $O(|\Delta\mathsf{C}|)$. □

In the rest of the chapter, we prove Theorem 2.3.4 by providing optimal algorithms for data that is partitioned vertically (Section 2.4) or horizontally (Sections 2.6).

2.4 Algorithms for Vertical Partitions

We start with an *optimal* incremental detection algorithm for vertical partitions $D = (D_1, \ldots, D_n)$. Here for $i \in [1,n]$, D_i resides at site S_i and $D_i = \pi_{X_i}(D)$ (see Section 2.2). The main result of this section is as follows.

Proposition 2.4.1: *There is an algorithm that incrementally detects CFD violations in vertical partitions with communication and computational costs in* $O(|\Delta D| + |\Delta\mathsf{V}|)$. □

It is nontrivial to develop an incremental detection algorithm bounded by $O(|\Delta D| + |\Delta\mathsf{V}|)$. To find $\Delta\mathsf{V}$, not only tuples in ΔD but also data in D may be needed and hence shipped. Indeed, as in Example 2.1.2, to validate ϕ_1 after t_6 is inserted into D_0 of Fig. 2.1, $t_5[\mathsf{street}, \mathsf{city}]$ in D_{V_2} and $t_5[\mathsf{CC}]$ in D_{V_3} are necessarily involved.

Below we shall first identify when the data in D is not needed in incremental detection. For the cases when the involvement of D is inevitable, we propose index structures to avoid shipping data in D. Based on the auxiliary structures, we then develop an optimal algorithm for vertically partitioned databases.

Cases independent of D. To validate a CFD $\phi = (X \to B, t_p)$ in response to the insertion or deletion of a tuple t, data in D is not needed in the following two cases.

(1) When ϕ is a *constant* CFD. Indeed, ϕ can be violated by a single tuple t alone. Hence to find $\Delta\mathsf{V}$ incurred by t, there is no need to consult other tuples in D.

(2) When ϕ is a variable CFD with $X \cup \{B\} \subseteq X_i$. In this case, ϕ can be *locally checked* at site S_i in which $D_i = \pi_{X_i}(D)$ resides. There is no need to ship data.

Index structures. Below we focus on validation of variable CFD $\phi = (X \to B, t_p)$, *i.e.,* $t_p[B] =$ '_'.

Observe that for a tuple t to make a violation of a CFD ϕ, there must exist some tuple t' such that $t[X] = t'[X]$, and moreover, either (a) $t[B] = t'[B]$ and t is already a violation of the CFD ϕ, or (b) $t[B] \neq t'[B]$, *i.e.,* $(t,t') \not\models \phi$. To capture this, we define an equivalence relation *w.r.t.* a set Y of attributes.

Equivalence classes. We say that tuples t and t' are *equivalent w.r.t.* Y if $t[Y] = t'[Y]$.

We denote by $[t]_Y$ the equivalence class of t, *i.e.,* $[t]_Y = \{t' \in D \mid t'[Y] = t[Y]\}$. We associate a unique identifier (eqid) $\mathsf{id}[t_Y]$ with $[t]_Y$.

We define a function eq() that takes as input the eqid's of equivalence classes $[t]_{Y_i}$ $(i \in [1,m])$, and returns the eqid of $[t]_Y$, where $Y = \bigcup_{i \in [1,m]} Y_m$, *i.e.,* $\mathsf{eq}(\mathsf{id}[t_{Y_1}], \cdots, \mathsf{id}[t_{Y_m}]) = \mathsf{id}[t_Y]$. As will be seen shortly, we send $\mathsf{id}[t_Y]$ rather than data in $[t]_{Y_i}$ to reduce the amount of data shipped.

Upon $[t]_Y$'s, we define the following index structures.

HEV-*index*. For each variable CFD $\phi = (X \to B, t_p)$, each sites S_i maintains a set of **H**ash-based **E**quivalence class and **V**alue indices (HEV's), denoted by HEV_i^ϕ. Each non-base HEV is a *key/value* store that given a tuple t and a set of eqid's $\mathsf{id}[t_{Y_j}]$ $(j \in [1,m])$ as the *key*, returns $\mathsf{id}[t_{Y_1 \cup \cdots \cup Y_m}]$ as the *value*. *Base* HEV's are also maintained to map distinct attribute values to their eqid's. These are special HEV's that take single attribute values as the key, and are shared by all CFDs. We write HEV_i for HEV_i^ϕ when ϕ is clear from the context.

Intuitively, HEV's help us identify $\mathsf{id}[t_X]$ and $\mathsf{id}[t_B]$, since all tuples that violate ϕ with t must be in $[t]_X$, and on attribute B, they have different values from $t[B]$.

The HEV's for CFD ϕ are organized as follows. We build HEV_X and HEV_B for attributes X and B, respectively. More specifically, we sort attributes of X into $(x_1, \ldots, x_m)$, and for each $i \in [1,m]$, we build an HEV for the subset $\{x_j \mid j \in [1,i]\}$. As will be seen in Example 2.4.1, to identify $\mathsf{id}[t_X]$, we use the HEV's for $\{x_1\}$, $\{x_1, x_2\}$, ..., $\{x_1, \ldots, x_m\}$ one by one in this order. We shall present the details of the strategy for building HEV's in Section 2.5, which aims to reduce eqid shipment when multiple CFDs are taken together.

IDX. We group tuples that violate ϕ with t into $[t']_{X \cup \{B\}}$ for each t' in $[t]_X$. The tuples are indexed by IDX, another hash index that is only stored at the site where $\mathsf{id}[t_X]$ is maintained. Given a tuple t, it returns a $\mathsf{set}(t[X])$ of distinct eqid's of $[t']_{X \cup \{B\}}$, where $t[X] = t'[X]$, and each eqid in turn identifies the set of all tuple ids in the equivalence class $[t']_{X \cup \{B\}}$. Intuitively, for each $[t]_X$, an IDX stores distinct values of B attribute and their associated tuple ids.

Example 2.4.1: Figure 2.3 depicts HEV's for ϕ_1 of Fig. 2.1 and relation D_0 of Fig. 2.2. HEV_2 and HEV_3 are the indices on sites S_2 and S_3, respectively, and the IDX is stored at S_2.

To compute $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$, we first find $\mathsf{id}[t_{5\{\mathsf{CC}\}}] = 1$ from a base hash table of HEV_3, since $t_5[\mathsf{CC}] = 44$, at site S_3. The eqid 1 (*i.e.,* $\mathsf{id}[t_{5\{\mathsf{CC}\}}]$) is then sent to S_2. Using

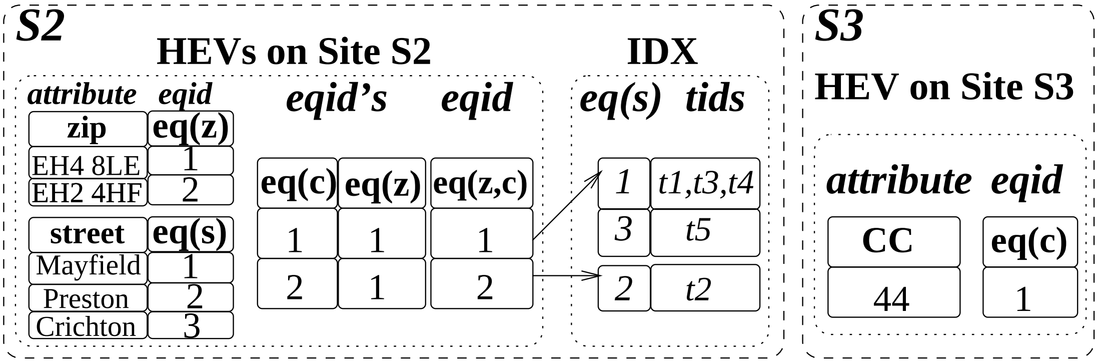


Figure 2.3: Example HEV-indices and an IDX for ϕ_1

the base hash table at site S_2, we get $\mathsf{id}[t_{5\{\mathsf{zip}\}}] = 1$ from $t_5[\mathsf{zip}] =$ EH4 8LE. Taking these together as the input for HEV_2, we get $\mathsf{eq}(1,1) = 1$, which is for $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$.

Moreover, as shown in Fig. 2.3, $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$ links to two entries in IDX, where 1 represents Mayfield with an equivalence class $\{t_1, t_3, t_4\}$, and 3 indicates Crichton with an equivalence class $\{t_5\}$.

Observe that during the detection, we use HEV's for eqid's of any tuple in this order: $\{\mathsf{CC}\}$ and $\{\mathsf{CC},\mathsf{zip}\}$. □

Example 2.4.1 tells us that to identify $\mathsf{id}[t_X]$, one only needs to ship at most $|X| - 1$ eqid's, to make the input for HEV_X, *i.e.,* the index of X.

Algorithms. Leveraging the index structures, we develop an incremental algorithm to detect violations in vertical partitions. To simplify the discussion, we first consider a single update for a single CFD. We then extend the algorithm to multiple CFDs and batch updates.

Single update for one CFD. Given a CFD ϕ, a vertically partitioned database D, violations $\mathsf{V}(\phi, D)$ of Σ in D, and a tuple t inserted into (resp. deleted from) D, the algorithm identifies changes $\Delta\mathsf{V}^+(\phi, D)$ (resp. $\Delta\mathsf{V}^-(\phi, D)$) to $\mathsf{V}(\phi, D)$. It first uses HEV to find the equivalence classes $[t]_X$ and its associate sets in IDX. It then computes $\Delta\mathsf{V}$.

Insertions. The algorithm for single-tuple insertion is shown in Fig. 2.4, referred to as incVIns. It first identifies $\mathsf{set}(t[X])$ by capitalizing on HEV-indices as discussed above (line 1). This requires to ship at most X eqid's, including the eqid of $t[B]$. When $|\mathsf{set}(t[X])| > 1$, all tuples t' such that (t', t) violate ϕ must have been found. Hence t is the only new violation (line 2; see Example 2.1.2). When $|\mathsf{set}(t[X])| = 1$, there are two cases: (1) if $\mathsf{set}(t[X])$ contains the entry for tuple t', where (t, t') violate ϕ, then t and all tuples in $[t']_{X\cup\{B\}}$ are new violations (line 4); and (2) if $\mathsf{set}(t[X])$ only contains

```
Algorithm incVIns
Input:   $\Delta D^+=\{t\}$, a vertically partitioned $D$, a variable CFD $\phi$
         and the old violations $\mathsf{V}(\phi, D)$.
Output:  $\Delta\mathsf{V}^+$.

    /* $\phi = (X \to B, t_p)$ */
1.  identify $\mathsf{set}(t[X])$ using HEV's and IDX's;
2.  if $|\mathsf{set}(t[X])| > 1$ then $\Delta\mathsf{V}^+ := \{t\}$;
3.  elseif $|\mathsf{set}(t[X])| = 1$ (i.e., $\mathsf{set}(t[X]) = \{t'\}$) then
4.      if $(t, t') \not\models \phi$ then $\Delta\mathsf{V}^+ := \{t\} \cup [t']_{X\cup\{B\}}$;
5.      else $\Delta\mathsf{V}^+ := \emptyset$;
6.  else $\Delta\mathsf{V}^+ := \emptyset$;
7.  augment IDX by adding $t$; HEV-indices are also maintained;
8.  return $\Delta\mathsf{V}^+$;

Algorithm incVDel
Input: $\Delta D^-=\{t\}$, a vertical partition $D$, a variable CFD $\phi$ and $\mathsf{V}(\phi, D)$.
Output: $\Delta\mathsf{V}^-$.
    /* $\phi = (X \to B, t_p)$ */
1.  identify $\mathsf{set}(t[X])$ and $[t]_{X\cup\{B\}}$ using HEV's and IDX's;
2.  if $|[t']_{X\cup\{B\}}| > 1$
3.      if $|\mathsf{set}(t[X])| > 1$ then $\Delta\mathsf{V}^- := \{t\}$;
4.      else $\Delta\mathsf{V}^- := \emptyset$;
5.  else    /* $|[t']_{X\cup\{B\}}| = 1$ */
6.      if $|\mathsf{set}(t[X])| > 2$ then $\Delta\mathsf{V}^- := \{t\}$;
7.      elseif $|\mathsf{set}(t[X])|=2$ (i.e., $\{t, t'\}$) then $\Delta\mathsf{V}^-:=\{t\} \cup [t']_{X\cup\{B\}}$;
8.      else $\Delta\mathsf{V}^- := \emptyset$;
9.  maintain IDX by deleting $t$; HEV-indices are also maintained;
10. return $\Delta\mathsf{V}^-$;
```

Figure 2.4: Single Insertion/Deletion for Vertical Partitions

the entry for t, then no violation arises (line 5). Otherwise, no tuple agrees with t on X attributes, and there is no violation (line 6). The new violations in $\Delta\mathsf{V}^+$ are then returned (line 8).

The index IDX is maintained in the same process, by inserting a tuple t into the set $[t]_{X\cup\{B\}}$, or adding an new entry to $\mathsf{set}(t[X])$ and its associated set $[t]_{X\cup\{B\}} = \{t\}$. In either case, it takes constant time. The HEV-indices are updated together with $\mathsf{id}[t_X]$. If such an eqid does not exist, a new entry is generated and added to the corresponding HEV-indices (line 7).

Deletions. The algorithm for single-tuple deletions, denoted as incVDel, is also shown in Fig. 2.4. It first finds both $[t]_{X\cup\{B\}}$ and $\mathsf{set}(t[X])$ using HEV (line 1). If no tuples are in $[t]_{X\cup\{B\}}$ after t is deleted (line 2), t is the only violation removed (line 3); otherwise there is no change to $V(\phi,D)$ (line 4). If t is the only tuple in $[t]_{X\cup\{B\}}$ (line 5), *i.e.,* the entry of t in $\mathsf{set}(t[X])$ will be removed, there are three cases to consider: (1) all violations *w.r.t.* t remain, and only t is removed (line 6); (2) all violations *w.r.t.* t are removed together with t when t is deleted (line 7); or (3) t does not violate ϕ (line 8). HEV and IDX indices are maintained similar to the case for insertions (line 9). Finally, $\Delta\mathsf{V}^-$ is returned (line 10).

Example 2.4.2: Consider D_0 (without t_6) of Fig. 2.2, ϕ_1 of Fig. 2.1, and its indices given in Fig. 2.3. When t_6 is inserted, at site S_3, it identifies $\mathsf{eq}(\mathsf{id}[t_{6\{\mathsf{CC}\}}]) = 1$ ($t_6[\mathsf{CC}] = 44$) from HEV$_3$ and ships this eqid (*i.e.,* 1) to S_2. At S_2, it identifies $\mathsf{eq}(\mathsf{id}[t_{6\{\mathsf{zip}\}}]) = 1$ ($t_6[\mathsf{CC}] =$ EH8 4LE) and $\mathsf{eq}(1,1) = 1$. This links to two entries in IDX as shown in Fig. 2.3, indicating that t_6 is the only new violation, *i.e.,* $\Delta\mathsf{V}^+ = \{t_6\}$ (line 2). Indeed, $\{t_5,t_6\} \not\models \phi_1$ and t_5 is a known violation. Only a single eqid (*i.e.,* 1) is shipped from site S_3 to site S_2.

Now suppose that tuple t_4 is deleted. Algorithm incVDel will find the eqid of $[t_4]_{\{\mathsf{CC,zip}\}}$ to be 1, which links to two entries, following the same process as above. After t_4 is deleted, $[t_4]_{\{\mathsf{CC,zip}\}}$ is not empty, *i.e.,* $[t_4]_{\{\mathsf{CC,zip}\}} = \{t_1,t_3\}$. Hence $\Delta\mathsf{V}^- = \{t_4\}$ (line 3). Again only a single eqid (*i.e.,* 1) is shipped. □

Batch updates and multiple CFDs. We now present an algorithm, denoted as incVer in Fig. 2.5, that takes *batch updates* ΔD, a vertically partitioned D, a set Σ of CFDs, and violations $\mathsf{V}(\Sigma,D)$ of Σ in D as input. It finds and returns the changes $\Delta\mathsf{V}$ of violations to $\mathsf{V}(\Sigma,D)$.

The algorithm works as follows. It first removes the updates in ΔD that cancel each other (line 1), and initializes the changes (line 2). It then detects the changes of violations for multiple CFDs in parallel (lines 3-16). It deals with three cases. (1) *Constant* CFDs (lines 4-10). It first identifies at each site S_i the tuple ids that can possibly match the pattern tuple t_p (line 5). These identified (partial) tuples are shipped to a designated coordinator site, together with corresponding B values (line 6). These tuple ids are naturally sorted in ascending order (by indices). A sort merge of them is thus conducted in linear time, and it generates a set T of tuples in which each tuple matches the

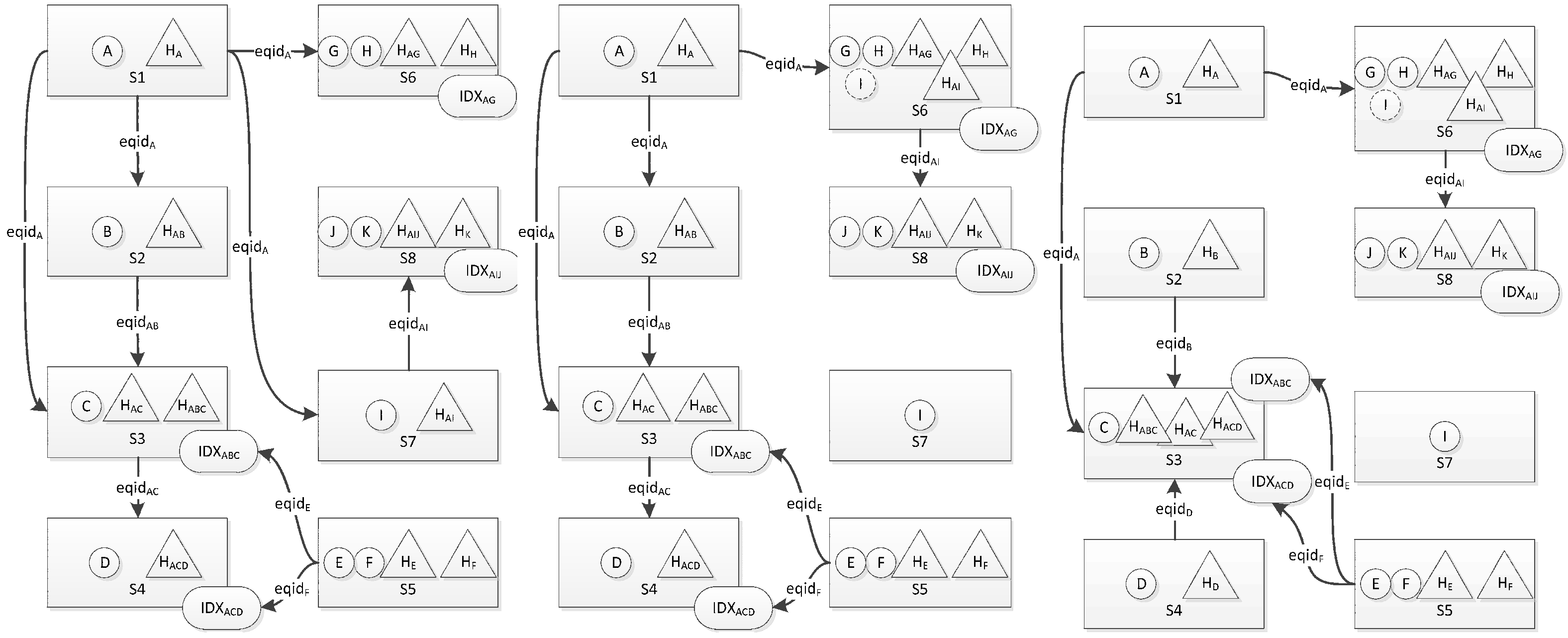
```
**Algorithm** incVer
*Input:*   $\Delta D$, $D$ in $n$ vertical partitions, a set $\Sigma$ of CFDs and $\mathsf{V}(\Sigma, D)$.
*Output:*  $\Delta\mathsf{V}$.
1.  remove updates in $\Delta D$ with the same tuple id and canceling each other;
2.  $\Delta\mathsf{V}^- := \emptyset$;   $\Delta\mathsf{V}^+ := \emptyset$;
3.  for each $\phi \in \Sigma$ do
4.    if $\phi$ is a constant CFD then /* $\phi = (X \to B, t_p)$ */
5.      $T_i := \{t \mid t \in \Delta D$ and $t[X_i \cap X] \asymp t_p[X_i \cap X]\}$ for $i \in [1, n]$;
6.      ship all $T_i$ with their values on $B$ attribute to one site;
7.      merge $T_i$ for $i \in [1, n]$ based on the same tuple id, get $T$;
8.      for each $t \in T$ do
9.        if $t[B] = t_p[B]$ and $t \in \Delta D^-$ then $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \{t\}$;
10.       elseif $t[B] \neq t_p[B]$ and $t \in \Delta D^+$ then $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \{t\}$;
11.   elseif $\phi$ can be locally checked at $S_i$ then
12.     derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ at $S_i$ use $\mathsf{HEV}_i$ and IDX (Section 2.4);
13.     $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$;   $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$
14.   else   /* a variable CFD that cannot be *locally checked* */
15.     derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ ($i \in [1, n]$) (see Fig. 2.4);
16.     $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$ and $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$ ($i \in [1, n]$);
17. return $\Delta\mathsf{V} = \Delta\mathsf{V}^- \cup \Delta\mathsf{V}^+$;
```

Figure 2.5: Batch Updates for Vertical Partitions

pattern tuple t_p on X attributes (line 7). It then examines these tuples' B attributes, to decide whether they are violations to be removed (line 9), or violations newly incurred (line 10). (2) *Locally checked variable* CFDs (lines 11-13). The changes of violations can be detected using the same indices as for a single CFD given above (lines 12-13). (3) *General variable* CFDs (lines 14-16). The method used is exactly what we have seen for a single CFD. The changes to violations are then returned (line 17).

Violations are marked with those CFDs that they violate when combining $\Delta\mathsf{V}$'s for multiple CFDs (see Fig. 2.1).

Complexity. For the communication cost, note that only eqid's are sent: for each tuple $t \in \Delta D$ and each CFD $\phi \in \Sigma$, its eqid's are sent at most $|X|$ times. As remarked earlier, the set Σ of CFDs and the fragmentation are fixed as commonly found in incremental integrity checking. Hence the messages sent are bounded by $O(|\Delta D|)$. The computational cost is in $O(|\Delta D| + |\Delta\mathsf{V}|)$, since checking both hash-based HEV and IDX take constant time, as well as their maintenance for each update.



(a) No replication

(b) With replication

(c) Minimum eqid shipment

Figure 2.6: Example of minimizing eqid shipment (base hash tables used only locally are omitted)

2.5 Optimization for Vertical Partitions

We have seen that by leveraging HEV's and IDX's, for vertical partition an incremental detection algorithm can be developed that is *bounded* in the changes in the input and output (*i.e.,* ΔD and ΔV). We next study how to build HEV's such that eqid shipment is minimized.

Recall that HEV's and IDX's are used together to identify the equivalent classes of the input update (line 1 of both algorithms incVIns and incVDel in Fig. 2.4), whilst for each variable CFD $(X \rightarrow B, t_p[X])$, two IDX's must be built with the key eqid_X and $\mathsf{eqid}_{X \cup \{B\}}$ respectively for each input tuple , and HEV's are built to efficiently compute these keys for IDX's. As remarked earlier, how these HEV's are built decides how eqid's are shipped for generating the keys of IDX's. For multiple CFDs that may have common attributes, different orders on grouping attributes of HEV's may affect the number of eqid's shipped for an single update, as shown below.

Example 2.5.1: Consider a relation R_e with 11 attributes $A, B, \cdots, K$ that is vertically partitioned and distributed over 8 sites: $S_1(A)$, $S_2(B)$, $S_3(C)$, $S_4(D)$, $S_5(E,F)$, $S_6(G,H)$, $S_7(I)$, $S_8(J,K)$. Here $S_1(A)$ denotes that attribute A is at site S_1 (besides a key); similarly for the other attributes. A set Σ_e of CFDs is imposed on R_e, including $\varphi_1 : (ABC \rightarrow E)$, $\varphi_2 : (ACD \rightarrow F)$, $\varphi_3 : (AG \rightarrow H)$, and $\varphi_4 : (AIJ \rightarrow K)$.

Consider different HEV's for the CFDs in Fig. 2.6, in which a rectangle indicates a site, a circle an attribute, a triangle an HEV, an ellipse an IDX index, and a directed edge indicates an eqid shipment from one site to another. Note that one IDX is needed for each CFD. We omit those base HEV's that only used locally to simplify the figure.

(1) *No sharing between the* HEV*'s of different CFDs.* Figure 2.6(a) depicts a case when HEV's are independently built for the CFDs. These HEV's determine how eqid's are shipped when validating the CFDs. For example, when a tuple t is inserted into (or deleted from) R_e, to detect the violations of $\varphi_1 : (ABC \rightarrow E)$, we need to (a) identify the eqid of $t[A]$ from H_A at site S_1, which is shipped to S_2; (b) determine the eqid of $t[AB]$ from H_{AB} upon receiving the eqid of $t[A]$, which is in turn shipped to S_3; (c) detect the new violations (resp. removed violations) for inserting (resp. deleting) t by examining H_{ABC} and the IDX index *w.r.t.* φ_1 at site S_3. Two eqid's need to be shipped for φ_1. The process for the other CFDs is similar. In total, 9 eqid's (*i.e.,* the number of directed edges in Fig. 2.6(a)) need to be shipped to detect all violations of the CFDs in Σ_e. Note that when the eqid of $t[A]$ is shipped from S_1 to S_3, it is used by both H_{AC} (for φ_2) and H_{ABC} (for φ_1) at site S_3; hence this eqid is shipped only once.

(2) *In the presence of replication.* Replication is common in distributed data management, to improve reliability and accessibility. Suppose that attribute I is replicated at site S_6 besides residing at S_7, as shown in Fig. 2.6(b). This allows us to choose either site S_6 or site S_7 where we build index H_{AI}, as opposed to Fig. 2.6(a) in which H_{AI} has to be built at S_7. Note that to detect the violations of $\varphi_3 : (AG \rightarrow H)$, the eqid for $t[A]$ needs to be shipped from S_1 to S_6 in both Fig. 2.6(a) and Fig. 2.6(b). If we build H_{AI} at S_6, we may send the eqid of $t[AI]$ from S_6 to S_8 (Fig. 2.6(b)), instead of from S_7 to S_8 (Fig. 2.6(a)) to validate $\varphi_4 = (AIJ \rightarrow K)$. This saves us one eqid shipment for $t[A]$ from S_1 to S_7 (Fig. 2.6(a)). In total, 8 eqid's need to be shipped in this case, instead of 9 in Fig. 2.6(a).

(3) *Sharing* HEV*'s among CFDs.* When I is replicated at site S_6, we can do better than Fig. 2.6(b), as depicted in Fig. 2.6(c). The key observation is that attributes AC are shared by CFDs φ_1 and φ_2. Hence, when a tuple t is inserted or deleted, we can compute the eqid of $t[AC]$ by shipping the eqid of $t[A]$ from S_1 to S_3. This allows us to compute the eqid's of $t[ABC]$ (with the eqid of $t[B]$ from S_2 to S_3) and $t[ACD]$ (with the eqid of $t[D]$ from S_4 to S_3) both at S_3 (Fig. 2.6(c)). In contrast, in the setting of Fig. 2.6(b) we have to compute eqid's by following the order of $t[A] \Rightarrow t[AB] \Rightarrow t[ABC]$ for φ_1 and $t[A] \Rightarrow t[AC] \Rightarrow t[ACD]$ for φ_2. In Fig. 2.6(c), only 7 eqid's need to be shipped as opposed to 8 eqid's in Fig. 2.6(b). □

Example 2.5.1 motivates us to find an optimal strategy for building HEV's, such that the keys of IDX's could be computed with minimum number of eqid shipments. It also suggests that we reduce eqid shipment by sharing HEV's among multiple CFDs as much as possible (*e.g.*, H_{AC} at S_3 for φ_1 and φ_2 in the case (3) above).

Below we first formalize this as an optimization problem, and show that it is NP-complete. We then provide an effective heuristic algorithm for building HEV's.

Optimization. A close look at the use of HEV in the detection algorithms and their complexity analysis (Section 2.4) reveals the following. To handle a unit update (insertion or deletion of a tuple t), the number of eqid's shipped is independent of (a) the values in database D and (b) the value of t. Indeed, eqid is shipped only when a non-base HEV needs eqid's generated from HEV's at other sites, and hence, is decided by the dependencies between HEV's. Thus we can talk about eqid shipments for a unit update regardless of the values of D and t.

We show that the problem of building HEV's is already challenging for unit updates. Consider a schema R, a vertical partition scheme that partitions an instance D of R into

$(D_1,\ldots,D_n)$ such that D_i resides at site S_i, and attributes of R may be replicated, *i.e.,* $(D_1,\ldots,D_n)$ may not be disjoint. Given a schema R, the partition and replication scheme for R, a set Σ of CFDs, and a positive number K, the *minimum* eqid *shipment problem* is to decide whether there exists a set $\mathcal{H}$ of HEV's such that for any instance D of R and any single update with tuple t, it needs no more than K eqid's shipped to find changes to $\mathsf{V}(\Sigma,D)$. Here for each $\varphi = (X \to B, t_p[X]) \in \Sigma$, $\mathcal{H}$ has to identify the keys eqid_X and $\mathsf{eqid}_{X\cup\{B\}}$ of two IDX's for φ, and it needs no more than K eqid shipments to find all such keys of IDX's for all CFDs in Σ.

Theorem 2.5.1: *The problem for minimum* eqid *shipment is NP-complete.* □

Proof. [‡] Upper bound. We show that the problem is in NP by giving an NP algorithm. It first guesses a set $\mathcal{H}$ of at most $\Sigma_{1\le i\le n}|R_i| + n*m$ hash tables with their locations, where $|R_i|$ is the number of attributes in partition D_i. Indeed, for each attribute in each D_i, one base hash table needs to be built (hence $\Sigma_{1\le i\le n}|R_i|$), and for each partition D_i and each CFD φ in Σ, we need at most 1 non-base hash table that contains all attributes of φ in D_i (hence $(m*n)$ non-base hash tables). After $\mathcal{H}$ is in place, we check (a) whether for any CFD $(X \to B, t_p[X]) \in \Sigma$, $\mathcal{H}$ can identify eqid_X and $\mathsf{eqid}_{X\cup\{B\}}$; and (b) whether we need no more than K eqid's shipped when validating all CFDs in Σ for a single update with tuple t. As remarked above, step (b) is independent of D and t. Steps (a) and (b) can be done by leveraging the dependencies between HEV's, in PTIME when the HEV's and their locations are given. If the number of eqid shipments is no more than K via $\mathcal{H}$, then $\mathcal{H}$ provides the indices we need. Otherwise we guess another $\mathcal{H}$ and repeat the process. This algorithm is in NP, and hence so is the problem.

Lower bound. We next show that problem is NP-hard by reduction from the minimum set cover problem (MSC; see the proof of Theorem 2.3.2 for the statement of MSC).

Given an instance $(X,\mathcal{C},K)$ of MSC, we construct (R,Σ,K) such that the minimum eqid shipment problem for (R,Σ,K) has a solution iff the MSC problem has a solution. Assume *w.l.o.g.* that $X = \{x_j \mid j \in [1,m]\}$, $\mathcal{C} = \{C_i \mid i \in [1,n]\}$, each C_i has three elements of X, and that $X{=}\bigcup_{i\in[1,n]} C_i$ (*i.e.,* there exists a cover for X).

(a) We define a schema $R = (\mathsf{id}, Y, Z, X_1, X_2, \ldots X_m)$, a partition and replication scheme that vertically partition any instance D of R into $n+1$ fragments $U, D_1, D_2, \ldots, D_n$, with schemas $R_U = (\mathsf{id}, Y)$ for U and $R_i = (\mathsf{id}, Z, X_{a_1}, X_{a_2}, X_{a_3})$ for D_i. Here x_{a_1}, x_{a_2} and x_{a_3} are elements in $C_i \in \mathcal{C}$. Intuitively, each D_i encodes a set C_i. and attributes may be

[‡] This proof is a joint work with Wenfei Fan.

duplicated in different sites.

(b) The set Σ consists of m FDs: $X_1Y \to Z$, $X_2Y \to Z$, $\ldots$, and $X_mY \to Z$. Intuitively, each $X_iY \to Z$ encodes the element x_i in X. Thus the set Σ encodes the set X.

We show that (R, Σ, K) is a reduction from MSC. First, assume that the MSC instance has a cover C' of size no larger than K. We define a set $\mathcal{H}$ as follows.

(a) On each site S_i, where $C_i = \{x_{a_1}, x_{a_2}, x_{a_3}\} \in C'$, $\mathcal{H}$ has the following HEV's: (i) $(h_{i0} : Z \to \mathsf{eqid}_Z)$; (ii) $(h_{i1} : X_{a_1} \to \mathsf{eqid}_{X_{a_1}})$, $(h_{i2} : X_{a_2} \to \mathsf{eqid}_{X_{a_2}})$, and $(h_{i3} : X_{a_1} \to \mathsf{eqid}_{X_{a_3}})$; (iii) $(h'_{i1} : \mathsf{eqid}_{X_{a_1}}, \mathsf{eqid}_Y \to \mathsf{eqid}_{X_{a_1}Y})$, $(h'_{i2} : \mathsf{eqid}_{X_{a_2}}, \mathsf{eqid}_Y \to \mathsf{eqid}_{X_{a_2}Y})$, and $(h'_{i3} : \mathsf{eqid}_{X_{a_3}}, \mathsf{eqid}_Y \to \mathsf{eqid}_{X_{a_3}Y})$; (iv) $(h''_{i1} : \mathsf{eqid}_{X_{a_1}}, \mathsf{eqid}_Y, \mathsf{eqid}_Z \to \mathsf{eqid}_{X_{a_1}YZ})$, $(h''_{i2} : \mathsf{eqid}_{X_{a_2}}, \mathsf{eqid}_Y, \mathsf{eqid}_Z \to \mathsf{eqid}_{X_{a_2}YZ})$, and $(h''_{i3} : \mathsf{eqid}_{X_{a_3}}, \mathsf{eqid}_Y, \mathsf{eqid}_Z \to \mathsf{eqid}_{X_{a_3}YZ})$.

(b) On the site S_U, $\mathcal{H}$ includes $(h_U : Y \to \mathsf{eqid}_Y)$.

Intuitively, to check a unit update t posed on any instance D of R, it suffices to ship the eqid_Y for t generated by (b) from S_U to S_i for each $C_i \in C'$. In total $|C'|$ eqid's are shipped (see the algorithms in Section 2.4). Indeed, since C' is a cover for X and Σ encodes X, one can verify the following: HEV's in (a)(iii) (resp. (a)(iv)) generate all eqid_{X_iY} (resp. eqid_{X_iYZ}) for each FD $(X_iY \to Z) \in \Sigma$, and all eqid's required for (a)(iii) and (a)(iv) are provided by eqid shipments of (c) for tuple t. Hence $\mathcal{H}$ suffices to generate all the eqid's needed by Σ. Since $|C'| \leq K$, the number of eqid shipments via $\mathcal{H}$ is at most K.

Conversely, assume that there exists a set $\mathcal{H}$ of hash tables such that for any FD $(X_iY \to Z) \in \Sigma$, $\mathcal{H}$ can find eqid_X and $\mathsf{eqid}_{X \cup \{B\}}$, and moreover, for any D and unit update with a tuple t, the number of eqid's shipped for computing eqid's of all CFDs in Σ is at most K. Consider the following cases. (a) If $K \geq n$, the set $\mathcal{C}$ is a cover and $|\mathcal{C}| = n \leq K$. (b) If $K < n$, let C' consist of those C_i's such that eqid's are shipped between U and D_i ($i \in [1, n]$) of $\mathcal{H}$ when handling the update. One can verify that $|C'| \leq K$ and C' is a cover for X, since otherwise, there must exist an uncovered element x_j in X such that eqid_{X_jY} for t could not be generated and checked. □

Due to the intractability, any efficient algorithm to find an optimal plan to build HEV's is necessarily heuristic.

A heuristic algorithm. We next provide an efficient heuristic algorithm for building HEV's. The idea behind the algorithm is to start with HEV's with the keys for IDX's. That is, for a CFD $\varphi = (X_\varphi \to Y_\varphi, t_{p_\varphi})$, we first build an HEV for X_φ, which is necessary for detecting violations of φ. We then build HEV's for certain subsets of X_φ, by selecting those subsets that contain as many attributes shared by multiple CFDs as possible. We also include base HEV's that contain attributes that only reside at one site, *e.g.,* H_A at site S_1 in Fig. 2.6(a), since H_{AB} at S_2 requires H_A at S_1 and local attribute B at S_2 as input, while H_{AB} at site S_2 in Fig. 2.6(a)) is not. Finally, we remove redundant HEV's while ensuring that all violations can still be detected. It follows a greedy approach that determines the key (set of eqid's) of each HEV and retains the HEV's with the minimum eqid shipment among the solutions explored. It terminates when no more HEV can be removed.

The algorithm, referred to as optVer, is shown in Fig. 2.7. It takes as input a database D that is vertically partitioned into D_i (for $i \in [1,n]$) and allows a predefined replication scheme, a set Σ of CFDs, and a parameter k for balancing the effectiveness and efficiency. It builds a set $\mathcal{H}$ of HEV's for Σ. The algorithm works as follows.

(1) [Initialization.] It builds a set $\mathcal{H}$ of HEV's such that for each $\varphi \in \Sigma$, there is an HEV with key X_φ (lines 1-4).

(2) [Expansion.] It then expands $\mathcal{H}$. For each CFD φ, we add up to $|\Sigma| + |X_\varphi|$ HEV's, by including the HEV's whose keys contain as many attributes shared by multiple CFDs as possible (lines 5-6). For each attribute of each CFD in Σ, we also build a base HEV (line 7), such that all existing HEV's can take their outputs and compute eqid's.

(3) [Location.] We assign a site to each HEV h in $\mathcal{H}$ (line 8). The site is determined by findLoc, such that (a) the local attributes at the site cover as many attributes of h as possible, and (b) as many other HEV's reside at the site as possible. This takes into account of the replication.

(4) [Finalization.] We follow a greedy approach to searching an optimal solution by removing HEV's from $\mathcal{H}$ (lines 9-18). After steps (2)–(4), some tables in $\mathcal{H}$ may be redundant, *i.e.,* unnecessary for computing those tables needed by IDX's ($\mathcal{H}_{\mathsf{IDX}}$). We iteratively remove HEV's from $\mathcal{H}$ until removing any more table will make some HEV in $\mathcal{H}_{\mathsf{IDX}}$ no longer computable (lines 10-18). In the process we record the best solution so far in $min_{\mathcal{H}}$ (line 13). More specifically, we conduct search in the BFS fashion: each state is a set of HEV's, Q keeps all open states, and the algorithm only includes the top k solutions (measured by the number of eqid shipped) in Q in each iteration (line 17), where k is a user defined threshold to balance the effectiveness and efficiency.

The function $\mathcal{H}.N_{\mathsf{eqid}}()$ computes the number of eqid shipments for a given set $\mathcal{H}$ of HEV's. It also determines the order and structure of each HEV h as follows: at each

```
Algorithm optVer
Input: $D$ in $n$ vertical partitions, a set $\Sigma$ of CFDs, a parameter $k$
Output: a set $min_{\mathcal{H}}$ of HEV's.
$\mathcal{H} := \emptyset$;
for each $\varphi \in \Sigma$ do            /* $\varphi : (X_\varphi \rightarrow Y_\varphi, t_{p_\varphi})$ */
  $\mathcal{H} := \mathcal{H} \cup \{$an HEV for $X_\varphi\}$;
$\mathcal{H}_{\mathsf{IDX}} := \mathcal{H}$; /* HEV's that are necessary for IDX's */
for each $\varphi \in \Sigma$ and $\phi \in \Sigma \setminus \{\varphi\}$ do $\mathcal{H} := \mathcal{H} \cup \{$an HEV for $X_\varphi \cap X_\phi\}$;
for each $\varphi \in \Sigma$ do add up to $|X_\varphi|$ HEV's having shared attributes;
Expand $\mathcal{H}$ with necessary base HEV's;
for each $h \in \mathcal{H}$ do $h$.location := findLoc($h$);
/* min and $min_{\mathcal{H}}$ keep the best solution so far; $\mathcal{H}.N_{\mathsf{eqid}}()$
returns #-eqid shipments for $\mathcal{H}$; $Q$ is the queue for BFS */
$min := \mathcal{H}.N_{\mathsf{eqid}}()$;   $min_{\mathcal{H}} := \mathcal{H}$;   $Q := \{\mathcal{H}\}$;
while ($Q \neq \emptyset$) do
  $Q' := \emptyset$;
  while ($\mathcal{H} = Q$.pop()) do
    if $min > \mathcal{H}.N_{\mathsf{eqid}}()$ then $min := \mathcal{H}.N_{\mathsf{eqid}}()$;   $min_{\mathcal{H}} := \mathcal{H}$;
    for each $h \in \mathcal{H}$ do
      if all HEV's in $\mathcal{H}_{\mathsf{IDX}}$ are computable by $(\mathcal{H} \setminus \{h\})$ then
        $Q'$.push($\mathcal{H} \setminus \{h\}$);
  Keep up to $k$ distinct $\mathcal{H}'$s with smallest $\mathcal{H}'.N_{\mathsf{eqid}}()$ in $Q'$;
  $Q := Q'$;
return $min_{\mathcal{H}}$;
```

Figure 2.7: Heuristic algorithm for minimizing eqid shipment

stage, it selects an HEV h' from $\mathcal{H}$ whose key attributes contain the largest number of uncovered attributes in h. The eqid computed from h' is to be shipped to h.

Example 2.5.2: Consider the data partition of Fig. 2.6(c) described in Example 2.5.1, where I is replicated at S_6. Taking these as input, optVer builds HEV's as follows.

(1) [Initialization.] It first builds 4 HEV's H_{ABC}, H_{ACD}, H_{AG} and H_{AIJ}, for CFDs φ_1, φ_2, φ_3, and φ_4, respectively.

(2) [Expansion.] It adds the following tables:

(a) H_A, since A is shared by all CFDs, and H_{AC}, as attributes AC are shared by φ_1 and φ_2;

(b) H_{AI} and H_{AJ}, in which keys are subsets of X_{φ_4}, and both contain attribute A; and

(c) base HEV for the CFDs in Σ_e: $H_B, \ldots, H_J, H_K$.

(3) [Location.] It assigns a site for each HEV to reside at: H_{ABC}, H_{ACD} at S_3, H_{AG} at S_6, and H_{AIJ} at S_8; each base HEV is located at the site where its attribute is located (*e.g.,* H_A at S_1 and H_B at S_2).

(4) [Finalization.] Assume that $k = 5$, it removes redundant H_{AJ}. The solution of Fig. 2.6(c) is then found, with 7 eqid's shipped in total. □

Complexity. The algorithm is in $O(k|\Sigma|^4 + n|\Sigma|)$ time. Indeed, it takes $O(k|\Sigma|^4)$ time for the iterations (lines 9–18) and $O(n|\Sigma|)$ time for site assignments (line 8). More specifically, the outer **while** iteration is bounded by the number of HEV's in $\mathcal{H}$ (*i.e.,* $O(|\Sigma|^2)$), the inner **while** iterates at most k times for each outer **while** iteration, the inner **for** loop runs at most $|\Sigma|^2$ times, and $N_{\mathsf{eqid}}()$ inside the **for** loop could be computed in $O(1)$ time using proper dynamic programming techniques. For other steps, it is in $O(|\Sigma|)$ time for lines 1-4, $O(|\Sigma|^2)$ time for line 5, and in $O(|\Sigma|^2)$ time for lines 6-7. Note that the number of rules $|\Sigma|$ is usually small in practice, and the algorithm only needs to be run once for given database D, replication scheme, and CFDs Σ instead of each time calling optVer at each update.

2.6 Algorithms for Horizontal Partitions

When it comes to horizontal partitions, there also exist incremental detection algorithms that are *optimal*.

Proposition 2.6.1: *There exists an algorithm that incrementally detects CFD violations in horizontal partitions with communication and computational costs in* $O(|\Delta D| + |\Delta \mathsf{V}|)$. □

Taken together, Propositions 2.4.1 and 2.6.1 verify Theorem 2.3.4.

Along the same lines as its vertical counterpart, we first identify when data shipment can be avoided. We then give an optimal algorithm for horizontal partitions.

Consider a database $D = (D_1, \ldots, D_n)$ that is horizontally partitioned, where D_i resides at site S_i for $i \in [1, n]$.

Local checking. For horizontal partitions, CFDs that can be validated locally include the following.

(1) Constant CFDs. Such a CFD can be violated by a *single* tuple, and does not incur global violations. Hence no data shipment is needed for validating constant CFDs.

(2) Variable CFDs. Notably, a horizontal fragment D_i is defined as $\sigma_{F_i}(D)$ (Section 2.2). We use X_{F_i} to denote all attributes in F_i. To validate a variable CFD $\phi = (X \to B, t_p)$, one does not have to ship data to or from S_i when

(a) $X_{F_i} \subseteq X$; indeed, for any tuple $t \in D_i$ and $t' \notin D_i$, (t,t') do not violate ϕ since $t[X_{F_i}] \neq t'[X_{F_i}]$; or

(b) $F_i \wedge F_\phi$ evaluates to false [FGMM10], where F_ϕ is a conjunction of atoms $A =$ 'a' imposed by t_p, for $A \in X$. Indeed, no tuples in D_i could possibly match $t_p[X]$.

Algorithms. We first consider a single CFD and a single update. We then extend the algorithm to multiple CFDs and batch updates. At each site, we also maintain the indices (*only* for local tuples) for equivalence classes and $\mathsf{set}()$ similar to the ones introduced in Section 2.4.

Single update for one CFD. Given a CFD $\phi = (X \to B, t_p)$ and a tuple t to be inserted into (resp. deleted from) D_i, the algorithm is to identify the changes $\Delta\mathsf{V}^+(\phi, D)$ (resp. $\Delta\mathsf{V}^-(\phi, D)$) to $\mathsf{V}(\varphi, D)$, outlined below.

Insertions. The algorithm handles insertions as follows.

(1) Site S_i checks local violations. It deals with two cases:

(a) There exist no local violations, *i.e.,* there is no $t' \in D_i$ such that $(t,t') \not\models \phi$. Then there are again two cases:

(i) when $[t]_{X\cup\{B\}} \neq \emptyset$: $\Delta\mathsf{V}_i^+ = \{t\}$ if $|\mathsf{set}(t[X])| > 1$, and $\Delta\mathsf{V}_i^+ = \emptyset$ otherwise; indeed, if $t' \in [t]_{X\cup\{B\}}$ is a known violation, so is t; or neither is a violation; and

(ii) when $[t]_{X\cup\{B\}} = \emptyset$: we need to send t to other sites to check global violations, *i.e.,* to find out whether there exists a tuple $t' \notin D_i$ such that $(t,t') \not\models \phi$. We set $\Delta\mathsf{V}_i^+ = \{t\}$ if such t' exists, and $\Delta\mathsf{V}_i^+ = \emptyset$ otherwise.

(b) Local violations exist, *i.e.,* there exists $t' \in D_i$ such that $(t,t') \not\models \phi$. We consider the following two cases:

(i) when $[t]_{X\cup\{B\}} \neq \emptyset$: then $\Delta\mathsf{V}_i^+ = \{t\}$, since any tuple that violates ϕ with t is a known violation; and

(ii) when $[t]_{X\cup\{B\}} = \emptyset$: then there must exist a tuple $t' \in D_i$ such that $(t,t') \not\models \phi$. If $t' \in \mathsf{V}_i$, we have $\Delta\mathsf{V}_i^+ = \{t\}$; otherwise $\Delta\mathsf{V}_i^+ = \{t\} \cup [t']_{X\cup\{B\}}$ since each tuple in $[t']_{X\cup\{B\}}$ violates ϕ with t. In both cases, we need to check global violations by sending t to all the other sites, which check violations incurred by inserting tuple t.

(2) Upon receiving t from S_i, each site S_j ($j \neq i$) checks its local violations *in parallel*, as described in step 1(a).

The global changes $\Delta\mathsf{V}^+$ is the union of changed violations from all the sites, *i.e.,* $\Delta\mathsf{V}^+ = \bigcup_{k \in [1,n]} \Delta\mathsf{V}_k^+$.

<u>*Deletions*</u>. When a tuple t is deleted from D_i at Site S_i, the algorithm does the following at S_i and other sites.

(1) *At site S_i*. It first identifies $[t]_{X \cup \{B\}}$ and $\mathsf{set}(t[X])$ at S_i for CFD φ. If t does not violate ϕ, then t is simply deleted from D_i, since deletions do not introduce new violations. When t violates ϕ, there are two cases to consider.

(a) If after t is deleted, tuples that agree with t on both X and B remain, then all violations except t remain.

(b) Otherwise, the entire entry for t will be removed. There are again two cases to consider:

(i) There are two items in $\mathsf{set}(t[X])$, t and t'. It broadcasts t' to the sites that have violations with t or t'. We record the sites that still have violations. It removes all violations *w.r.t.* t and t' if no sites have tuples that violate t', and otherwise only t is removed from violations.

(ii) Tuple t is the only entry at site S_i. It removes t as a violation, and broadcasts t to the other sites that previously have violations with t.

The local index is maintained and $\Delta\mathsf{V}_i^-$ is then returned.

(2) At site S_j. Upon receiving t from S_i, each site S_j ($j \neq i$) checks whether previous violations maintained at S_j could be removed. Note that S_j will send two different messages: either (a) t' from S_i ((1)(b)(i) above): this means that t' remains at S_i; or (b) t from S_i ((1)(b)(ii) above): this means that t is removed from S_i.

The global changes $\Delta\mathsf{V}^-$ is the union of $\Delta\mathsf{V}_k^-$ ($k = [1,n]$), from all individual sites.

Example 2.6.1: Consider D_0 (without t_6) given in Fig. 2.2 and ϕ_1 of Fig. 2.1. When tuple t_6 is inserted, the algorithm finds that $(t_6, t_5) \not\models \phi_1$ at site S_3 (step (1)(a)), *i.e.,* no local violations. However, since t_5 is a known violation (Fig. 2.1), so is t_6 (step (1)(a)(i)). Hence, $\Delta\mathsf{V}^+ = \{t_6\}$. □

```
Algorithm incHor
Input:   $\Delta D$, $D$ in $n$ horizontal partitions, $\Sigma$, and $\mathsf{V}(\Sigma, D)$.
Output:  $\Delta\mathsf{V}$.
1.  merge local updates in $\Delta D_i$ having the same tuple ids;
2.  $\Delta\mathsf{V}^- := \emptyset$;   $\Delta\mathsf{V}^+ := \emptyset$;
3.  for each $\phi \in \Sigma$ do
4.    if $\phi$ is a constant CFD then /* $\phi = (X \rightarrow B, t_p)$ */
5.      for each $t \in \Delta D_i$ $(i \in [1,n])$ and $t$ violates $\phi$ do
6.        if $t \in \Delta D_i^-$ then $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \{t\}$;
7.        elseif $t \in \Delta D_i^+$ then $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \{t\}$;
8.    elseif $\phi$ can be locally checked at $S_i$ then
9.      derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ at $S_i$ with indices (Section 2.6);
10.     $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$;   $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$
11.   else   /* a variable CFD that cannot be locally checked */
12.     derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ $(i \in [1,n])$;
13.     $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$ and $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$ $(i \in [1,n])$;
14. return $\Delta\mathsf{V} = \Delta\mathsf{V}^- \cup \Delta\mathsf{V}^+$;
```

Figure 2.8: Batch updates for horizontal partitions

Batch updates and multiple CFDs. We now present an algorithm for batch updates and multiple CFDs on horizontal partitions, denoted as incHor and shown in Fig. 2.8. Given batch updates ΔD, a horizontal partition $(D_1, \cdots, D_n)$ of a database D, a set Σ of CFDs, and (old) violations $\mathsf{V}(\Sigma, D)$ of Σ in D, the algorithm finds and outputs the changes $\Delta\mathsf{V}$ to violations $\mathsf{V}(\Sigma, D)$.

The algorithm first removes the local updates that cancel each other (line 1), and initializes the changes (line 2). It then detects the changes to violations for multiple CFDs in parallel (lines 3-13). It deals with three cases as follows. (1) *Constant* CFDs (lines 4-7). It checks at each site that whether a deletion removes a violation (line 6) or an insertion adds a violation (line 7). (2) *Locally checked variable* CFDs (lines 8-10). The changes to violations can be detected using the same indices as used in Section 2.4, in constant time (lines 9-10). (3) *General variable* CFDs (lines 11-13). The changes to violations are identified (lines 12-13), and then returned (line 14).

Complexity. For communication cost, one can see that each tuple in ΔD is sent to other sites at most once. Hence at most $O(|\Delta D|\ n)$ messages are sent, where n is the number of fragments and is fixed, as remarked earlier. Thus the cost is in $O(|\Delta D|)$. The computation cost is in $O(|\Sigma|(|\Delta D| + |\Delta V|))$ time, where $|\Sigma|$ is a fixed parameter. That is, it is in $O(|\Delta D| + |\Delta V|)$. Indeed, by leveraging hash tables, the process at each site takes constant time, and the hash tables can be maintained incrementally in the same process, also in constant time.

Optimization using MD5. A tuple may be large. To reduce its shipping cost, a natural idea is to encode the whole tuple, and then send the coding of the tuple instead of the tuple. MD5 (Message-Digest algorithm 5 [Wik12]) is a widely used cryptographic hash function with a 128-bit hash value. We use MD5 in our implementation to further reduce the communication cost, by sending a 128-bit MD5 code instead of an entire tuple.

2.7 Experimental Study

We present an experimental study of our incremental algorithms for vertical and horizontal partitions, evaluating elapsed time and data shipment. We focus on their scalability by varying four parameters: (1) $|D|$: the size of the base relation; (2) $|\Delta D|$: the size of updates; (3) $|\Sigma|$: the number of CFDs; and (4) n: the number of partitions. We also evaluated the effectiveness of our optimization techniques for building indices in vertical partitions.

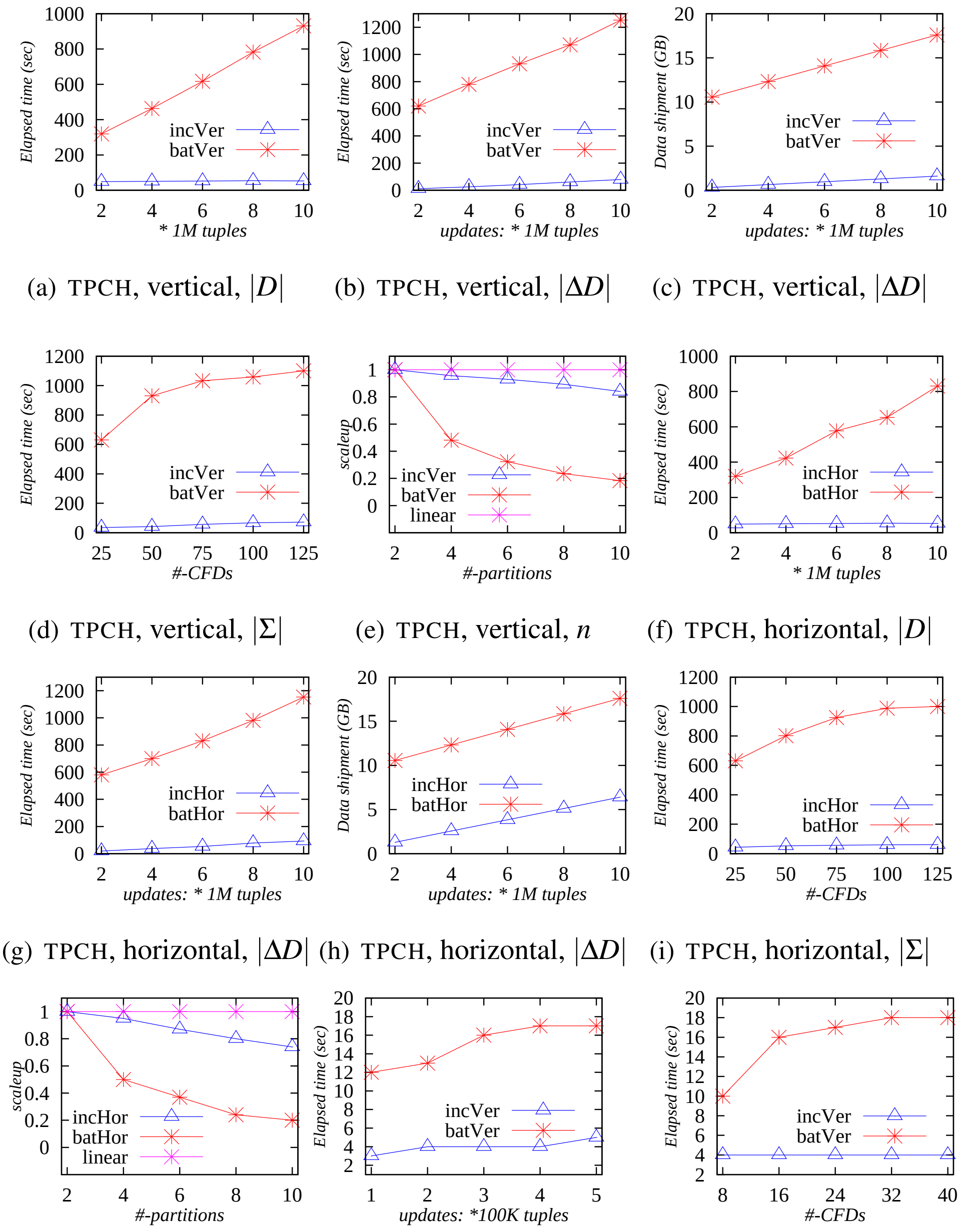
Experimental setting. We used the following datasets.

(1) Datasets. (a) TPCH: we joined all tables to build one table. The data ranges from 2 million tuples (*i.e.,* 2M) to 10 million tuples (*i.e.,* 10M). Notably, the size of 10M tuples is **10GB**. (b) DBLP: we extracted a 320MB relation from its XML data. It scales from 100K to 500K tuples.

(2) CFDs were designed manually. We first designed functional dependencies (FDs), and then produced CFDs by adding patterns (*i.e.,* conditions) to the FDs. For TPCH: the number $|\Sigma|$ of CFDs ranges from 25 to 125, with increment of 25 by default. For DBLP: $|\Sigma|$ scales from 8 to 40, with increment of 8 by default.

(3) Updates. Batch updates contain 80% insertions and 20% deletions, since insertions happen more often than deletions in practice. The size of updates is up to 10M tuples (about **10GB**) for TPCH and up to 320MB for DBLP.

(4) Partitions. Its fragment number is 10 by default.



(a) TPCH, vertical, $|D|$ (b) TPCH, vertical, $|\Delta D|$ (c) TPCH, vertical, $|\Delta D|$

(d) TPCH, vertical, $|\Sigma|$ (e) TPCH, vertical, n (f) TPCH, horizontal, $|D|$

(g) TPCH, horizontal, $|\Delta D|$ (h) TPCH, horizontal, $|\Delta D|$ (i) TPCH, horizontal, $|\Sigma|$

(j) TPCH, horizontal, n (k) DBLP, vertical, $|\Delta D|$ (l) DBLP, vertical, $|\Sigma|$

Figure 2.9: Experimental results for TPCH and DBLP data

Implementation. We denote by incVer (resp. incHor) our incremental algorithms for batch updates and multiple CFDs in vertical (resp. horizontal) partitions. We also designed batch algorithms for detecting errors in vertical (resp. horizontal) partitions, denoted by batVer (resp. batHor), following [FGMM10]. The batch algorithms work in three steps: (1) for each CFD it copies to a coordinator site a small number of relevant attributes (resp. tuples) for vertical (resp. horizontal) partitions; (2) the violations of each CFD ϕ are checked locally at the coordinator site for ϕ; and (3) the violations of all CFDs are checked in parallel. All algorithms were written in Python. We ran our experiments on Amazon EC2 High-Memory Extra Large instances (zone: us-east-1c).

In the following, we shall pay more attention to TPCH, more interesting for its larger size than DBLP.

Experimental results for vertical partitions. We first present our experimental results of detecting violations in data that is vertically partitioned and distributed.

Exp-1: Impact of $|D|$. Fixing $|\Delta D| = 6$M, $|\Sigma| = 50$ and $n = 10$, we varied the size of D (*i.e.,* $|D|$) from 2M to 10M tuples (10GB) for TPCH. Figure 2.9(a) shows the elapsed time in seconds when varying $|D|$. The result tells us that incVer outperforms batVer by two orders of magnitude. It also shows that the elapsed time of incVer is insensitive to $|D|$. In contrast, the elapsed time of batVer increases much faster when $|D|$ is increased. This result further verifies Proposition 2.4.1: the incremental algorithm is bounded by the size of the changes in the input and output, and it is independent of D.

Exp-2: Impact of $|\Delta D|$. Fixing $|\Sigma| = 50$, $n = 10$ and $|D|$ = 10M, we varied the size of ΔD from 2M to 10M tuples for TPCH. We also varied $|\Delta D|$ from 100K to 500K tuples for DBLP while fixing $|D| = 500$K, $|\Sigma| = 16$ and $n = 10$.

Figure 2.9(b) (resp. Figure 2.9(k)) shows the elapsed time in seconds when varying $|\Delta D|$ for TPCH (resp. DBLP). Both figures show that the elapsed time of incVer increases almost linearly with $|\Delta D|$, *e.g.,* 11 seconds when $|\Delta D| = 2$M and 79 seconds when $|\Delta D| = 10$M as shown in Fig. 2.9(b). In addition, batVer is slower than incVer by two orders of magnitude, consistent with Fig. 2.9(a).

In addition, Figure 2.9(c) shows the size of data shipped (in GB) when varying $|\Delta D|$ for TPCH. Note that incVer only sends 320MB when $|\Delta D| = 2$M (*i.e.,* 2GB) and 1.6GB when $|\Delta D| = 10$M (*i.e.,* 10GB). This is because with HEVs, we only ship eqid's instead of the entire tuples. In contrast, the size of data shipped for batVer is up to 17.6GB when $|\Delta D|$ = 10M. This further verifies our observation from Figure 2.9(b).

These experimental results tell us that our incremental methods are bounded by $|\Delta D| + |\Delta V|$, independent of the size of D, in contrast to batch algorithms that detect violations starting from scratch, which depends on $|D|$.

Exp-3: Impact of $|\Sigma|$. Fixing n=10, $|D|$=10M and $|\Delta D|$=6M for TPCH, we varied $|\Sigma|$ from 25 to 125. Fixing n=10, $|D|$=500K and $|\Delta D|$=300K for DBLP, we varied $|\Sigma|$ from 8 to 40. Figure 2.9(d) (resp. Figure 2.9(l)) shows the elapsed time when varying $|\Sigma|$ from 25 to 125 for TPCH (resp. from 8 to 40 for DBLP). Both figures show that incVer

Dataset	**without optimization** #-eqid **shipments**	**with optimization** #-eqid **shipments**
TPCH	122	55
DBLP	61	17

Figure 2.10: Number of eqid's shipped for vertical partitions

achieves almost linear scalability when varying $|\Sigma|$, *e.g.,* 35 seconds when $|\Sigma|$=25 and 72 seconds when $|\Sigma|$=125 in Fig. 2.9(d). When multiple CFDs are detected, multiple sites work in parallel to improve the efficiency. Moreover, batVer runs far slower than incVer, as expected.

The results demonstrate that incVer scale well with $|\Sigma|$, and it can handle a large number of CFDs. We remark that in practice, Σ is typically predefined and fixed.

Exp-4: Impact of n. In this set of experiments, we varied the number of partitions from 2 to 10, and varied $|D|$ and $|\Delta D|$ in the same scale correspondingly. That is, we varied both $|D|$ and $|\Delta D|$ from 2M to 10M for TPCH. We study the *scaleup* performance defined as follows:

$$\text{scaleup} = \frac{\text{small system elapsed time on small problem}}{\text{large system elapsed time on large problem}}$$

Scaleup is said to be *linear* if it is 1, the ideal case.

Figure 2.9(e) shows the scaleup performance when varying n, $|D|$ and $|\Delta D|$ at the same time, where x-axis represents n and y-axis the scaleup value. The line for *linear* is the ideal case. For example, we computed the scaleup when $n = 4$ as follows: using the elapsed time when $n = 2$ and $|D| = |\Delta D| = 2$M to divide the elapsed time when $n = 4$ and $|D| = |\Delta D| = 4$M tuples (*i.e.,* 4GB in size), which is 0.96; similarly for all the other points. This figure shows that incVer achieves nearly *linear* scaleup, which clearly outperforms batVer that shows bad scaleup performance.

These results indicate that incVer scales well with partitions, when base data and updates are large.

Optimization for vertical partitions. We next evaluate the effectiveness of our optimization strategy (Section 2.5).

Exp-5. Figure 2.10 shows the number of eqid's shipped for vertically partitioned TPCH ($D = 10$M, $|\Sigma| = 50$, and $n = 10$) and DBLP ($D = 500$K, $|\Sigma| = 16$, and $n = 10$), with or without using the optimization methods presented in Section 2.5. As remarked earlier, for each tuple insertion or deletion, the amount of eqid's shipped is independent of $|D|$. The table tells us that for both datasets, the optimization technique significantly reduces the number of eqid's to be shipped: it saves 67 eqid's (55.5%) for TPCH and 44 eqid's (72.1%) for DBLP per update.

Experimental results for horizontal partitions for TPCH. We next present results on horizontally partitioned data.

Exp-6: Impact of $|D|$. We adopted the same setting as Exp-1. Figure 2.9(f) shows the elapsed time when varying $|D|$. Besides telling us that incHor outperforms batHor, the results also show that incHor is independent of D: when varying $|D|$ from $2M$ to $10M$ tuples, the time only changes slightly. This verifies Proposition 2.6.1: incremental violation detection in horizontal partitions depends only on $|\Delta D|$ and $|\Delta \mathsf{V}|$, and is independent of D.

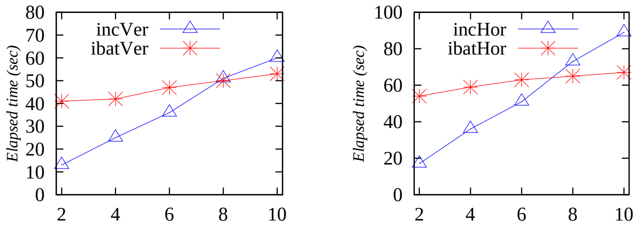
Exp-7: Impact of $|\Delta D|$. We used the same setting as Exp-2. Figure 2.9(g) shows the elapsed time when varying $|\Delta D|$ for TPCH. The results show that incHor increases almost linearly with the size of ΔD, *e.g.,* 19 seconds when $|\Delta D| = 2M$ and 93 seconds when $|\Delta D| = 10M$. Figure 2.9(h) shows the size of data shipment for both methods. The results verify that our incremental detection algorithm for horizontal partitions is bounded by $|\Delta D|$, similar to its vertical counterpart (see Exp-2).

Exp-8: Impact of $|\Sigma|$. We adopted the same setting as Exp-3. Figure 2.9(i) shows the elapsed time when varying $|\Sigma|$ from 25 to 125. It tells us that incHor is almost linear in $|\Sigma|$, *e.g.,* 43 seconds when $|\Sigma| = 25$ and 61 seconds when $|\Sigma| = 125$. The results verify that incHor scales well with $|\Sigma|$, as its vertical counterpart (see Exp-3).

Exp-9: Impact of n. Figure 2.9(j) shows the scaleup performance of incHor when varying n, $|D|$ and $|\Delta D|$ in the same scale, where x-axis represents the number n of fragments and y-axis the scaleup values. From the results we can see that incHor has nearly ideal scaleup, as its vertical counterpart. This verifies that our algorithms can work well on massive data, updates, and partitions.

Exp-10. Algorithms incVer and incHor substantially outperform existing batch algorithms. To favor the batch approach, we improved the batch algorithms, denoted by ibatVer and ibatHor for vertical and horizontal partitions, respectively, by using our incremental insertion algorithms and indices. We evaluated the performance of incVer and incHor vs. ibatVer and ibatHor starting with $\emptyset$, and inserting and deleting tuples until it reaches D.

Figure 2.11(a) (resp. Figure 2.11(b)) shows the result for vertical (resp. horizontal) partition when $|D| = 6\text{M}$, $|\Sigma| = 50$ and $n = 10$, while varying $|\Delta D|$ from 2M to 10M with 40% deletions and 60% insertions. The performance of batVer and batHor is not shown, since they are two orders of magnitude slower. The results tell us that in both vertical and horizontal partitions, the incremental algorithms do better than the revised batch algorithms until updates ΔD get rather large, *e.g.*, $|\Delta D|$ = 8M for vertical partitions and 7.6M for horizontal partitions.



(a) **Ver**, $|\Delta D|$: $*$1M tuples (b) **Hor**, $|\Delta D|$: $*$1M tuples

Figure 2.11: Experimental results for refined batch algorithms

Summary. From the experimental results we find the following. (1) Our incremental algorithms scale well with $|D|$, $|\Delta D|$ and $|\Sigma|$ for both vertical partitions (Exp-1 to Exp-4) and horizontal partitions (Exp-6 to Exp-9). (2) The incremental algorithms outperform their batch counterparts by two orders of magnitude, for reasonably large updates. But when updates are very large, batch algorithms do better, as expected (Exp-10). (3) The optimization techniques of Section 2.5 substantially reduce data shipment for vertical partitions (Exp-5). We contend that these incremental methods are promising in detecting inconsistencies in large-scale distributed data, for both vertically and horizontally partitioned data.

Chapter 3

Towards Certain Fixes with Editing Rules and Master Data

3.1 Introduction

In this chapter, we study the problem of finding certain fixes by data monitoring which is related the other task of data consistency: data repairing.

Data monitoring is to find errors in t and correct the errors, when a tuple t is committed to a database (either manually generated by users or automatically by programs. That is, we want to make sure that t is cleaned before it is committed to the database, to prevent errors introduced by adding t. As noted by [SMO07], it is far less costly to correct t at the point of data entry than fixing it afterward.

Integrity constraints have been studied for decades. Many are used for data cleaning, from traditional constraints (*e.g.,* functional and inclusion dependencies [BFFR05, CM05, Wij05]) to their extensions (*e.g.,* conditional functional and inclusion dependencies [FGJK08, BFM07, GKK$^+$08]). With the help of these constraints, we can determine whether data is dirty or not, *i.e.,* whether errors are present in the data. However, integrity constraints fall short of identifying which attributes of t are erroneous and moreover, how to correct the errors.

Example 3.1.1: Consider an input tuple t_1 given in Fig. 3.1(a). It specifies a supplier in the UK in terms of name (FN, LN), phone number (area code AC and phone phn) and type, address (street str, city, zip code) and items supplied. Here phn is either home phone or mobile phone, indicated by type (1 or 2, respectively).

It is known that in the UK, if AC is 020, city should be Ldn, and when AC is 131,

	FN	LN	AC	phn	type	str	city	zip	item
t_1:	Bob	Brady	020	079172485	2	501 Elm St.	Edi	EH7 4AH	CD
t_2:	Robert	Brady	131	6884563	1	null	Ldn	null	CD
t_3:	Robert	Brady	020	6884563	1	null	null	EH7 4AH	DVD
t_4:	Mary	Burn	029	9978543	1	null	Cad	null	BOOK

(a) Example input tuples t_1, t_2, t_3 and t_4

	FN	LN	AC	Hphn	Mphn	str	city	zip	DOB	gender
s_1:	Robert	Brady	131	6884563	079172485	51 Elm Row	Edi	EH7 4AH	11/11/55	M
s_2:	Mark	Smith	020	6884563	075568485	20 Baker St.	Ldn	NW1 6XE	25/12/67	M

(b) Example master relation D_m

Figure 3.1: Example input tuples and master relation

city must be Edi. These can be expressed as CFDs [FGJK08]. The CFDs find that tuple t_1 is *inconsistent*: $t_1[\mathsf{AC}] = 020$ but $t_1[\mathsf{city}] = \text{Edi}$. In other words, either $t_1[\mathsf{AC}]$ or $t_1[\mathsf{city}]$ is incorrect, or both. However, they do not tell us which of the two attributes is wrong and to what value it should be changed. □

Many methods have been studied for repairing data based on constraints [ABC03, BFFR05, CFG+07, FH76, KL09, HSW09b]. Most are heuristics. For the reasons mentioned above, however, these methods can not guarantee to generate correct fixes; worse still, they may introduce new errors when trying to repair the data. For instance, the tuple s_1 of Fig. 3.1(b) indicates corrections to t_1. Nevertheless, the prior methods may opt to change $t_1[\mathsf{city}]$ to Ldn; this does not fix the erroneous $t_1[\mathsf{AC}]$ and worse, messes up the correct attribute $t_1[\mathsf{city}]$.

This highlights the quest for effective methods to find *certain fixes* that are guaranteed correct [Gil88, HSW09b]. The need for this is especially evident when it comes to *critical* data, in which a seemingly minor error may have disastrous consequences [HSW09b]. To this end, we propose *editing rules* that tell us how to fix errors, *i.e.,* which attributes are wrong and what values they should take. In contrast, constraints only detect the presence of errors.

This is possible given the recent development of master data management (MDM [RW08]). An enterprise nowadays typically maintains *master data* (*a.k.a. reference data*), a single repository of high-quality data that provides various applications with a synchronized, consistent view of its core business entities. MDM systems are being developed by IBM, SAP, Microsoft and Oracle. In particular, master data has been explored to provide *a data entry solution* in the Service Oriented Architecture (SOA) at IBM [SMO07], for data monitoring.

Example 3.1.2: A master relation D_m is shown in Fig. 3.1(b). Each tuple in D_m specifies a person in the UK in terms of the name (FN, LN), home phone (Hphn), mobile phone (Mphn), address, date of birth (DOB) and gender. An example editing rule eR_1 is:

- ○ for an input tuple t, if there exists a master tuple s in D_m with $s[\mathsf{zip}] = t[\mathsf{zip}]$, then t should be updated by $t[\mathsf{AC}, \mathsf{str}, \mathsf{city}] := s[\mathsf{AC}, \mathsf{str}, \mathsf{city}]$, provided that $t[\mathsf{zip}]$ is *certain*, *i.e.,* it is assured correct by the users.

This rule makes *corrections* to attributes $t[\mathsf{AC}]$, $t[\mathsf{str}]$ and $t[\mathsf{city}]$, by taking values from the master tuple s_1.

Another editing rule eR_2 is:

- ○ if $t[\mathsf{type}] = 2$ (indicating mobile phone) and if there is a master tuple s with $s[\mathsf{Mphn}] = t[\mathsf{phn}]$, then $t[\mathsf{FN}, \mathsf{LN}] := s[\mathsf{FN}, \mathsf{LN}]$, as long as $t[\mathsf{phn}, \mathsf{type}]$ is certain.

This *standardizes* $t_1[\mathsf{FN}]$ by changing Bob to Robert.

As another example, consider tuple t_2 in Fig. 3.1(a), in which $t_2[\mathsf{str}, \mathsf{zip}]$ are missing, and $t_2[\mathsf{AC}]$ and $t_2[\mathsf{city}]$ are inconsistent. Consider an editing rule eR_3:

- ○ if $t[\mathsf{type}] = 1$ (indicating home phone) and if there exists a master tuple s in D_m such that $s[\mathsf{AC}, \mathsf{phn}] = t[\mathsf{AC}, \mathsf{Hphn}]$, then $t[\mathsf{str}, \mathsf{city}, \mathsf{zip}] := s[\mathsf{str}, \mathsf{city}, \mathsf{zip}]$, provided that $t[\mathsf{type}, \mathsf{AC}, \mathsf{phn}]$ is *certain*.

This helps us fix $t_2[\mathsf{city}]$ and *enrich* $t_2[\mathsf{str}, \mathsf{zip}]$ by taking the corresponding values from the master tuple s_1. □

Related work. The work of this chapter was originally published in [FLM$^+$12]. This chapter contains part of the content also included in [Ma11], including examples, notions etc. for keeping the integrity and completeness of this chapter. This work extends [FLM$^+$10] by including (1) a comprehensive analysis of the fundamental problems in connection with certain fixes (Section 3.4); (2) an interactive framework and algorithm for finding certain fixes (Section 3.5), and (3) its experimental study (Section 3.7). Neither (2) nor (3) was studied in [FLM$^+$10]. All the proofs and some of the results of (1) were not presented in [FLM$^+$10]. Due to the space constraint we opt to cover these new results by leaving out the deduction algorithms for certain regions and their experimental study of [FLM$^+$10].

A variety of constraints have been studied for data cleaning, such as FDs [Wij05], FDs and inclusion dependencies (INDs) [BFFR05], CFDs [CFG$^+$07, FGJK08], conditional inclusion dependencies (CINDs) [BFM07], matching dependencies (MDs) [FGJ$^+$11], and extensions of CFDs and CINDs [BFGM08, CFM09] (see *e.g.,* [Fan08] for a survey). (a) These constraints help us determine whether data is dirty or not, but they do not tell us which attributes are erroneous or how to fix the errors, as illustrated earlier. (b) The static analyses of those constraints have been focusing on the satisfiability and implication problems [FGJK08, BFM07, FGJ$^+$11, BFGM08, CFM09], along the same lines as traditional FDs and INDs [AHV95]. Editing rules differ from those constraints in the following: (a) they are defined in terms of updates, and (b) their reasoning is relative to master data and is based on its dynamic semantics, a departure from our familiar terrain of dependency analysis. The rules aim to fix errors, rather than to detect the presence of errors only.

Editing rules are also quite different from edits studied for census data repairing [FH76, Gil88, HSW09b]. Edits (a) are conditions defined on single records of a single relation, and (b) are not capable of locating and fixing errors.

Closer to editing rules are MDs [FGJ$^+$11]. In contrast to editing rules, (a) MDs are for record matching (see *e.g.,* [EIV07] for a survey), not for data repairing. (b) They only specify what attributes should be identified, but do not tell us how to update them. (c) MDs neither carry data patterns, nor consider master data; and hence, their analysis is far less challenging. Indeed, the static analyses are in PTIME for MDs [FGJ$^+$11], but in contrast, the analyses are intractable for editing rules.

There has also been work on rules for active databases (see [WC96] for a survey). Those rules are far more general than editing rules, specifying events, conditions and actions. Indeed, even the termination problem for those rules is undecidable, as opposed to the coNP upper bounds for editing rules. Results on those rules do not carry over to editing rules.

Prior work on constraint-based data cleaning has mostly focused on two topics introduced in [ABC03]: *repairing* is to find another consistent database that minimally differs from the original database [ABC03, BFFR05, RH01, CM05, BFM07, FGJK08, FH76, Gil88, HSW09b, KL09, YEN$^+$11, CCC$^+$10]; and *consistent query answering* is to find an answer to a given query in every possible repair of the original database (*e.g.,* [ABC03, Wij05]). Although the need for finding certain fixes has long been recognized [Gil88, HSW09b], prior methods do not guarantee that fixes are correct, *i.e.,* new errors may be introduced while fixing existing ones in the repairing process. Moreover, master data is not considered in those methods. We shall evaluate the effectiveness of our approach compared with the repairing algorithm of [CFG$^+$07] (Section 3.7).

This work studies data monitoring, which is advocated in [CGGM03, FPS$^+$10, SMO07, CCC$^+$10], as opposed to prior data repairing methods [ABC03, BFFR05, CM05, BFM07, FGJK08, FH76, Gil88, HSW09b, KL09, Wij05] that aim to generate another database as a candidate repair of the original data. As noted by [SMO07], it is far less costly to correct t at the point of entry than fixing it afterward. A method for matching input tuples with master data was presented in [CGGM03], without repairing the tuples.

Another line of work on data cleaning has focused on record matching [FH76, FGJ$^+$11, BGMM$^+$09, GDSZ10], to identify records that refer to the same real-world object (see [EIV07] for a survey). This work involves record matching between input tuples and master tuples. There has also been a host of work on more general data cleaning and ETL tools (see [BS06] for a survey), which are essentially orthogonal, but complementary, to data repairing and this work.

There have also been efforts to interleave merging and matching operations [NBBW06, FLM$^+$11a, BGMM$^+$09, GDSZ10]: [GDSZ10] clusters data rather than repair data, and [BGMM$^+$09, GDSZ10] only merge/fuse tuples when matches are found. Those merge operations are far more restrictive than value modifications considered in this work and data repairing. While [FLM$^+$11a] conducts both repairing and matching using CFDs and MDs, these operations cannot assure the correctness of the repaired data. Indeed, the prior work neither guarantees certain fixes, nor considers master data.

Our data monitoring framework leverages user feedback, similar to [RH01, YEN$^+$11, CCC$^+$10]. Potter's Wheel [RH01] supports interactive data transformations, based on iterative user feedback on example data. USHER [CCC$^+$10] cleans data by asking users online about erroneous values, identified by a probabilistic method. GDR [YEN$^+$11] develops a CFD-based repairing approach by soliciting user feedback on the updates that are likely to improve data quality. Our approach asks users to assure the correctness of a small number of attributes for an input tuple, to find a certain fix. While all these methods interact with users, they differ from each other in what feedback is requested and how the feedback is used.

Editing rules can be extracted from business rules. They can also be automatically discovered from sample data along the same lines as mining constraints for data cleaning, *e.g.,* [CM08, GKK$^+$08] for CFDs and [SC09] for MDs.

Related work. The work of this chapter was originally published in [FLM$^+$12]. This chapter contains part of the content also included in [Ma11], including examples, notions etc. for keeping the integrity and completeness of this chapter. This work extends [FLM$^+$10] by including (1) a comprehensive analysis of the fundamental problems in connection with certain fixes (Section 3.4); (2) an interactive framework and algorithm for finding certain fixes (Section 3.5), and (3) its experimental study (Section 3.7). Neither (2) nor (3) was studied in [FLM$^+$10]. All the proofs and some of the results of (1) were not presented in [FLM$^+$10]. Due to the space constraint we opt to cover these new results by leaving out the deduction algorithms for certain regions and their experimental study of [FLM$^+$10].

Organization. Section 3.2 defines editing rules. Section 3.3 presents certain fixes. Section 3.4 studies fundamental problems in connection with certain fixes. An interactive framework for data monitoring is introduced in Section 3.5. The experimental study is presented in Section 3.7.

3.2 Editing Rules

We study editing rules for data monitoring. Given a master relation D_m and an input tuple t, we want to fix errors in t using editing rules and data values in D_m.

We specify input tuples t with a relation schema R, and use $A \in R$ to denote that A is an attribute of R. The master relation D_m is an instance of a relation schema R_m, often distinct from R. As remarked earlier, D_m can be assumed consistent and complete [RW08].

Editing rules. An *editing rule* [FLM$^+$12] (eR) φ defined on (R,R_m) is a pair $((X,X_m) \rightarrow (B,B_m),t_p[X_p])$, where

- ○ X and X_m are two lists of distinct attributes in schemas R and R_m, respectively, with the same length, *i.e.,* $|X| = |X_m|$;
- ○ B is an attribute such that $B \in R \setminus X$, and attribute $B_m \in R_m$; and
- ○ t_p is a pattern tuple over a set of distinct attributes X_p in R such that for each $A \in X_p$, $t_p[A]$ is one of _, a or $\bar{a}$. Here a is a constant drawn from the domain of A, and _ is an unnamed variable.

Intuitively, a and $\bar{a}$ specify Boolean conditions $x = a$ and $x \neq a$ for a value x, respectively, and _ is a wildcard that imposes no conditions. More specifically, we say that a tuple t of R *matches* pattern tuple t_p, denoted by $t[X_p] \approx t_p[X_p]$, if for each attribute $A \in X_p$, (1) $t[A] = a$ if $t_p[A]$ is a, (2) $t[A] \neq a$ if $t_p[A]$ is $\bar{a}$, and (3) $t[A]$ is any value from the domain of A if $t_p[A]$ is _.

Example 3.2.1: Consider the supplier schema R and master relation schema R_m shown in Fig. 3.1(b). The rules eR$_1$, eR$_2$ and eR$_3$ described in Example 3.1.2 can be expressed as the following editing rules φ_1– φ_4 defined on (R,R_m).

φ_1: $((\mathsf{zip},\mathsf{zip}) \to (B_1,B_1), t_{p1} = ())$;

φ_2: $((\mathsf{phn},\mathsf{Mphn}) \to (B_2,B_2), t_{p2}[\mathsf{type}] = (2))$;

φ_3: $(([\mathsf{AC},\mathsf{phn}],[\mathsf{AC},\mathsf{Hphn}]) \to (B_3,B_3), t_{p3}[\mathsf{type},\mathsf{AC}]$
$= (1, \overline{0800}))$;

φ_4: $((\mathsf{AC},\mathsf{AC}) \to (\mathsf{city},\mathsf{city}), t_{p4}[\mathsf{AC}] = (\overline{0800}))$.

Here eR_1 is expressed as three editing rules of the form φ_1, for B_1 ranging over $\{\mathsf{AC}, \mathsf{str},\mathsf{city}\}$. In φ_1, both X and X_m consist of zip, and B and B_m are B_1. Its pattern tuple t_{p1} poses no constraints. Similarly, eR_2 is expressed as two editing rules of the form φ_2, in which B_2 is either FN or LN. The pattern tuple $t_{p2}[\mathsf{type}] = (2)$, requiring that phn is mobile phone. The rule eR_3 is written as φ_3 for B_3 ranging over $\{\mathsf{str},\mathsf{city},\mathsf{zip}\}$, where $t_{p3}[\mathsf{type},\mathsf{AC}]$ requires that type = 1 (home phone) yet AC $\neq$ 0800 (toll free, non-geographic). The eR φ_4 states that for a tuple t, if $t[\mathsf{AC}] \neq 0800$ and $t[\mathsf{AC}]$ is correct, we can update $t[\mathsf{city}]$ using the master data. □ □

Semantics. We next introduce the semantics of editing rules [FLM+12].

We say that an eR φ and a master tuple $t_m \in D_m$ *apply* to an R tuple t, which results in a tuple t', denoted by $t \to_{(\varphi,t_m)} t'$, if (1) $t[X_p] \approx t_p[X_p]$, (2) $t[X] = t_m[X_m]$, and (3) t' is obtained by the update $t[B] := t_m[B_m]$. We shall simply say that (φ,t_m) apply to t.

That is, if t matches t_p and if $t[X]$ agrees with $t_m[X_m]$, then we assign $t_m[B_m]$ to $t[B]$. Intuitively, if $t[X,X_p]$ is assured correct (referred to as *validated*), we can safely *enrich* $t[B]$ with master data $t_m[B_m]$ as long as (1) $t[X]$ and $t_m[X_m]$ are identified, and (2) $t[X_p]$ matches the pattern in φ. This yields a new tuple t' such that $t'[B] = t_m[B_m]$ and $t'[R \setminus \{B\}] = t[R \setminus \{B\}]$.

We write $t \to_{(\varphi,t_m)} t$ if φ and t_m do *not* apply to t, *i.e.*, t is unchanged by φ if either $t[X_p] \not\approx t_p[X_p]$ or $t[X] \neq t_m[X_m]$.

Example 3.2.2: As shown in Example 3.1.2, we can correct t_1 by applying the eR φ_1 and master tuple s_1 to t_1. As a result, $t_1[\mathsf{AC},\mathsf{str}]$ is changed from (020, 501 Elm St.) to (131, 51 Elm Row). Furthermore, we can standardize $t_1[\text{FN}]$ by applying φ_2 and s_1 to t_1, such that $t_1[\text{FN}]$ is changed from Bob to Robert.

The eR φ_3 and master tuple s_1 can be applied to t_2, to correct $t_2[\mathsf{city}]$ and enrich $t_2[\mathsf{str},\mathsf{zip}]$. □ □

Notations. Following [FLM+12], we use the following notations.

(1) Given an eR $\varphi = ((X,X_m) \to (B,B_m),t_p[X_p])$, we denote (a) $\text{LHS}(\varphi) = X$, $\text{RHS}(\varphi) = B$; (b) $\text{LHS}_m(\varphi) = X_m$, $\text{RHS}_m(\varphi) = B_m$; and (c) $\text{LHS}_p(\varphi) = X_p$.

(2) Given a set Σ of eRs, we denote $\cup_{\varphi\in\Sigma}\text{LHS}(\varphi)$ by $\text{LHS}(\Sigma)$; similarly for $\text{RHS}(\Sigma)$, $\text{LHS}_m(\Sigma)$ and $\text{RHS}_m(\Sigma)$. Here abusing the notions for sets, we use $X \cup Y$, $X \cap Y$ and $X \setminus Y$ to denote the *union*, *intersection* and *difference* of two lists X and Y of attributes, respectively.

(3) An eR $\varphi = ((X, X_m) \rightarrow (B, B_m), t_p[X_p])$ is said to be in *the normal form* if $t_p[X_p]$ does not contain wildcard $\_$. Every eR φ can be normalized to an eR φ' by removing all such attributes A from $t_p[X_p]$ that $t_p[A] = \_$. From the semantics of eRs one can readily verify that φ and φ' are *equivalent*: for any input tuple t, master tuple t_m, and tuple t', $t \rightarrow_{(\varphi,t_m)} t'$ iff $t \rightarrow_{(\varphi',t_m)} t'$.

Remarks. (1) As remarked earlier, editing rules are quite different from CFDs [FGJK08]. A CFD $\psi = (X \rightarrow Y, t_p)$ is defined on a single relation R, where $X \rightarrow Y$ is a standard FD and t_p is a pattern tuple on X and Y. It requires that for any tuples t_1, t_2 of R, if t_1 and t_2 match t_p, then $X \rightarrow Y$ is enforced on t_1 and t_2. When $t_p[Y]$ consists of constants only, it is referred to as a constant CFD. It has a *static* semantics: t_1 and t_2 either satisfy or violate ψ, but they are not updated. As shown in Example 3.1.1, when t_1 and t_2 violate φ, one cannot tell which of $t_1[X]$, $t_1[Y]$ or $t_2[Y]$ is erroneous, and hence, cannot simply apply φ to find a certain fix. The problem remains even when φ is a constant CFDs, which can be violated by a single tuple. In contrast, an eR φ specifies an action: applying φ and a master tuple t_m to t yields an updated t'. It is defined in terms of master data. As will be seen shortly, this yields a certain fix when φ and t_m are applied to a region that is validated.

(2) MDs of [FGJ$^+$11] also have a dynamic semantics. An MD ϕ is of the form $((X, X'), (Y, Y'), \mathsf{OP})$, where X, Y and X', Y' are lists of attributes in schemas R, R', respectively, and OP is a list of similarity operators. For an R_1 tuple t_1 and an R_2 tuple t_2, ϕ states that if $t_1[X]$ and $t_2[X']$ match *w.r.t.* the operators in OP, then $t_1[Y]$ and $t_2[Y']$ are identified as the same object. As remarked in Section 3.1, eRs differ from MDs in several aspects.

Neither CFDs nor MDs are expressible as eRs, and *vice versa*, because of their different semantics.

(3) To simplify the discussion we consider a single master relation D_m. Nonetheless the results of this work readily carry over to multiple master relations. Indeed, given master schemas $R_{m_1}, \ldots, R_{m_k}$, there exists a single master schema R_m such that each instance D_m of R_m characterizes an instance of $(D_{m_1}, \ldots, D_{m_k})$ of those schemas. Here

R	Input relation schema
R_m	Master relation schema
Σ	A set of eRs on (R, R_m)
D_m	Master data on R_m
$\bar{a}$	Boolean condition $x \neq a$ for a value x
$t \approx t_c$	An input tuple t *matches* a pattern tuple t_c
$t \rightarrow_{(\varphi, t_m)} t'$	Applying eR φ and a master tuple t_m to an input tuple t, yielding t'

Table 3.1: Summary of notations of Section 3.2

R_m has a special attribute id such that $\sigma_{\mathsf{id}=\mathsf{i}}(R_m)$ yields D_{m_i} for $i \in [1, k]$.

We summarize notations of this section in Table 3.1.

3.3 Certain Fixes and Certain Regions

Consider a master relation D_m of schema R_m, and a set Σ of editing rules defined on (R, R_m). Given a tuple t of R, we want to find a "certain fix" t' of t by using Σ and D_m. That is, (1) no matter how eRs of Σ and master tuples in D_m are applied, Σ and D_m yield a unique t' by updating t; and (2) all the attributes of t' are ensured correct (validated).

To formalize the notion of certain fixes, we first introduce a notion of regions. When applying an eR φ and a master tuple t_m to t, we update t with values in t_m. To ensure that the changes make sense, some attributes of t have to be validated. In addition, we are not able to update t if either it does not match the pattern tuple of φ or it cannot find a master tuple t_m in D_m that carries the information needed for correcting t.

Example 3.3.1: Consider the master data D_m of Fig. 3.1(b) and a set Σ_0 consisting of $\varphi_1, \varphi_2, \varphi_3$ and φ_4 of Example 3.2.1. Both (φ_1, s_1) and (φ_3, s_2) apply to tuple t_3 of Fig. 3.1(a). However, they suggest to update t_3[city] with distinct values Edi and Lnd. The conflict arises because t_3[AC] and t_3[zip] are inconsistent. Hence to fix t_3, we need to assure that one of t_3[AC] and t_3[zip] is correct.

Now consider tuple t_4 of Fig. 3.1(a). Since no eRs in Σ_0 and master tuples in D_m can be applied to t_4, we cannot tell whether t_4 is correct. This is because Σ_0 and D_m do not cover all the cases of input tuples. □ □

This motivates us to introduce the following notion.

Regions. A *region* is a pair (Z, T_c), where Z is a list of distinct attributes in R, T_c is a *pattern tableau* consisting of a set of pattern tuples with attributes in Z, and each pattern tuple is defined as its counterparts in eRs.

We say that a tuple t is *marked* by (Z, T_c) if there exists $t_c \in T_c$ such that $t \asymp t_c$.

Intuitively, a region (Z, T_c) species what input tuples can be corrected with certain fixes by a set Σ of eRs and master data. As will be seen shortly, (1) it tells us that to correctly fix errors in a tuple t, $t[Z]$ should be assured correct, and moreover, t is marked such that there exist an eR and a master tuple that can be applied to t. (2) There exist no two eRs in Σ such that both of them can be applied to t, but they lead to inconsistent updates. In other words, T_c imposes constraints *stronger than* those specified by pattern tuples in eRs, to prevent the abnormal cases illustrated in Example 3.3.1.

Consider an eR $\varphi = ((X, X_m) \rightarrow (B, B_m), t_p[X_p])$, a master tuple t_m and a region (Z, T_c). When we apply φ and t_m to a tuple t marked by (Z, T_c), we require that $X \subseteq Z$, $X_p \subseteq Z$, $B \notin Z$. That is, it is justified to apply φ and t_m to t for those t marked by (Z, T_c) if $t[X, X_p]$ is correct. As $t[Z]$ is validated, we make $t[B]$ "protected", *i.e.,* unchanged, by enforcing $B \notin Z$. We denote this as $t \rightarrow_{((Z,T_c),\varphi,t_m)} t'$, where $t \rightarrow_{(\varphi,t_m)} t'$.

Example 3.3.2: Referring to Example 3.3.1, a region defined on R is $(Z_{\mathsf{AH}}, T_{\mathsf{AH}})$ = $((\mathsf{AC}, \mathsf{phn}, \mathsf{type}), \{(\overline{0800}, \_, 1)\})$. Note that tuple t_3 of Fig. 3.1(a) is marked by $(Z_{\mathsf{AH}}, T_{\mathsf{AH}})$. Hence, if $t_3[\mathsf{AC}, \mathsf{phn}, \mathsf{type}]$ is validated, then (φ_3, s_2) can be applied to t_3, yielding $t_3 \rightarrow_{((Z_{\mathsf{AH}},T_{\mathsf{AH}}),\varphi_3,s_2)} t_3'$, where $t_3'[\mathsf{str}, \mathsf{city}, \mathsf{zip}] := s_2[\mathsf{str}, \mathsf{city}, \mathsf{zip}]$, and t_3' and t_3 agree on all the other attributes of R. □ □

Note that if $t \rightarrow_{((Z,T_c),\varphi,t_m)} t'$, then $t'[B]$ is validated as a logical consequence of the application of φ and t_m, since $t[Z]$ is validated. That is, $t'[B]$ is assured correct when applying rules to t' in the process for fixing t (see below). Hence we can extend (Z, T_c) by including B in Z and by expanding each t_c in T_c such that $t_c[B] = \_$. We denote the extended region as $\mathsf{ext}(Z, T_c, \varphi)$.

Example 3.3.3: Consider the region $(Z_{\mathsf{AH}}, T_{\mathsf{AH}})$ in Example 3.3.2. Then $\mathsf{ext}(Z_{\mathsf{AH}}, T_{\mathsf{AH}}, \varphi_3)$ is (Z', T'), where Z' consists of attributes $\mathsf{AC}, \mathsf{phn}, \mathsf{type}$, str, city and zip, and T' has a single pattern tuple t_c' = $(\overline{0800}, \_, 1, \_, \_, \_)$. □ □

Fixes. We say that a tuple t' is a *fix* of t by (Σ, D_m) *w.r.t.* (Z, T_c), denoted by $t \rightarrow^{*}_{((Z,T_c),\Sigma,D_m)} t'$, if there exists a finite sequence $t_0 = t$, t_1, …, $t_k = t'$ of tuples of

R such that for each $i \in [1,k]$, there exist $\varphi_i \in \Sigma$ and $t_{m_i} \in D_m$ such that

(1) $t_{i-1} \rightarrow_{((Z_{i-1},T_{i-1}),\varphi_i,t_{m_i})} t_i$, where $(Z_0,T_0) = (Z, T_c)$ and $(Z_i, T_i) = \mathsf{ext}(Z_{i-1},T_{i-1},\varphi_i)$; and

(2) for all $\varphi \in \Sigma$ and $t_m \in D_m$, $t' \rightarrow_{((Z_k,T_k),\varphi,t_m)} t'$.

These conditions ensure that (1) each step of the process is justified; and (2) t' is a fixpoint and cannot be further updated. Note that $t_{i-1} \rightarrow_{((Z_{i-1},T_{i-1}),\varphi_i,t_{m_i})} t_i$ assures that $t_i[Z] = t_0[Z] = t[Z]$, *i.e.,* $t[Z]$ is assumed correct and hence, remains unchanged in the process.

Unique fixes. We say that an R tuple t has a *unique fix* by (Σ, D_m) *w.r.t.* (Z,T_c) if there exists a unique t' such that $t \rightarrow^*_{((Z,T_c),\Sigma,D_m)} t'$. When there exists a unique fix t' of t with a finite sequence $t_0 = t$, t_1, ..., $t_k = t'$ of tuples of R, we refer to Z_k as the set of attributes of t *covered by* (Z,T_c,Σ,D_m).

Certain fixes. We say that an R tuple t has a *certain fix* by (Σ, D_m) *w.r.t.* (Z,T_c) if (1) t has a unique fix and (2) the set of attributes covered by (Z,T_c,Σ,D_m) includes *all* the attributes in R.

A notion of deterministic fixes was addressed in [Gil88, HSW09b]. It refers to unique fixes, *i.e.,* (1) above, without requiring (2). Further, it is not defined relative to (Z,T_c).

Intuitively, a unique fix t' becomes a certain fix when the set of attributes covered by (Z,T_c,Σ,D_m) includes *all* the attributes in R. We can find a certain fix for a tuple t of R marked by a region (Z,T_c) if (a) $t[Z]$ is assured correct, (b) there is a unique fix t'; and (c) all the remaining values of $t'[R \setminus Z]$ are correctly fixed.

Example 3.3.4: By the set Σ_0 of eRs of Example 3.3.1 and the master data D_m of Fig. 3.1(b), tuple t_3 of Fig. 3.1(a) has a unique fix *w.r.t.* $(Z_{\mathsf{AH}}, T_{\mathsf{AH}})$, namely, t_3' given in Example 3.3.2. However, as observed in Example 3.3.1, if we extend the region by adding zip, denoted by $(Z_{\mathsf{AHZ}}, T_{\mathsf{AH}})$, then t_3 no longer has a unique fix by (Σ_0, D_m) *w.r.t.* $(Z_{\mathsf{AHZ}}, T_{\mathsf{AH}})$.

As another example, consider a region $(Z_{\mathsf{zm}}, T_{\mathsf{zm}})$, where $Z_{\mathsf{zm}} = (\mathsf{zip}, \mathsf{phn}, \mathsf{type})$, and T_{zm} has a single tuple $(\_,\_,2)$. As shown in Example 3.2.2, tuple t_1 of Fig. 3.1(a) has a unique fix by Σ_0 and D_m *w.r.t.* $(Z_{\mathsf{zm}}, T_{\mathsf{zm}})$, by correctly applying (φ_1, s_1) and (φ_2, s_2). It is *not* a certain fix, since the set of attributes covered by $(Z_{\mathsf{zm}}, T_{\mathsf{zm}}, \Sigma_0, D_m)$ does not include item. Indeed, the master data D_m of Fig. 3.1(b) has no information about item, and hence, does not help here. To find a certain fix, one has to extend Z_{zm} by adding item. In other words, its correctness has to be assured by the users. □ □

(Z,T_c)	A *region* with a list Z of distinct attributes and a pattern tableau T_c
$t \rightarrow_{((Z,T_c),\varphi,t_m)} t'$	*Applying* eR φ and master tuple t_m to input tuple t *w.r.t.* (Z,T_c), yielding t'
$t \rightarrow^{*}_{((Z,T_c),\Sigma,D_m)} t'$	Tuple t' is a *fix* of input tuple t by (Σ, D_m) *w.r.t.* (Z,T_c)
attributes *covered*	All those attributes in t' that are validated by $t \rightarrow^{*}_{((Z,T_c),\Sigma,D_m)} t'$

Table 3.2: Summary of notations of Section 3.3

Certain regions. We next introduce the last notion of this section. We say that a region (Z,T_c) is a *certain region* for (Σ,D_m) if for all tuples t of R that are marked by (Z,T_c), t has a certain fix by (Σ,D_m) *w.r.t.* (Z,T_c).

We are naturally interested in certain regions since they warrant absolute corrections, which are assured either by the users (the attributes Z) or by master data (the remaining attributes $R \setminus Z$).

Example 3.3.5: As shown in Example 3.3.4, $(Z_{\mathsf{zm}},T_{\mathsf{zm}})$ is not a certain region. One can verify that a certain region for (Σ_0,D_m) is $(Z_{\mathsf{zmi}},T_{\mathsf{zmi}})$, where Z_{zmi} extends Z_{zm} by including item, and T_{zmi} consists of patterns of the form $(z,p,2,\_)$ for z,p ranging over $s[\mathsf{zip},\mathsf{Mphn}]$ for all master tuples s in D_m. For those tuples marked by the region, certain fixes are warranted.

Another certain region for (Σ_0,D_m) is $(Z_{\mathsf{L}},T_{\mathsf{L}})$, where $Z_{\mathsf{L}} = (\mathrm{FN},\mathrm{LN},\mathsf{AC},\mathsf{phn},\mathsf{type},\mathsf{item})$, T_{L} consists of pattern tuples of the form $(f,l,a,h,1,\_)$, and (f,l,a,h) is $s[\mathrm{FN},\mathrm{LN},\mathsf{AC},\mathsf{Hphn}]$ for all $s \in D_m$. □ □

We summarize notations in Table 3.2.

3.4 Static Analyses of Fundamental Problems

Given a set Σ of eRs and a master relation D_m, we want to make sure that they can *correctly* fix *all* errors in those input tuples marked by a region (Z,T_c). This motivates us to study fundamental problems associated with certain fixes by (Σ, D_m) and (Z,T_c), and establish their complexity and approximation bounds.

3.4.1 Reasoning about Editing Rules

We start with the problems for reasoning about editing rules when regions are provided. Given (Σ, D_m) and a region (Z, T_c), we want to know (a) whether (Σ, D_m) and (Z, T_c) have any conflicts when put together (referred to as the consistency problem), and (b) whether (Z, T_c) makes a certain region for (Σ, D_m) (known as the coverage problem). We show that these problems are intractable, but identify PTIME special cases.

The consistency problem. We say that (Σ, D_m) is *consistent relative to* (Z, T_c) if for each input R tuple t marked by (Z, T_c), t has a unique fix by (Σ, D_m) *w.r.t.* (Z, T_c). Intuitively, this says that Σ and D_m do not have conflicts *w.r.t.* (Z, T_c), as illustrated below.

Example 3.4.1: There exist (Σ, D_m) and (Z, T_c) that are inconsistent. Indeed, (Σ_0, D_m) described in Example 3.3.1 is not consistent relative to region $(Z_{\mathsf{AHZ}}, T_{\mathsf{AHZ}})$ of Example 3.3.4, since eRs in Σ_0 suggest distinct values to update $t_3[\mathsf{city}]$ for tuple t_3 of Fig. 3.1(a), *i.e.,* conflicts arise, as shown in Example 3.3.1. Hence t_3 does not have a unique fix by (Σ_0, D_m) *w.r.t.* $(Z_{\mathsf{AHZ}}, T_{\mathsf{AHZ}})$. □ □

The *consistency problem* for editing rules is to determine, given any (Z, T_c) and (Σ, D_m), whether (Σ, D_m) is consistent relative to (Z, T_c).

The problem is obviously important, but is nontrivial. It is known that for constraints defined with pattern tuples, the presence of attributes with a finite domain makes their static analysis hard [BFM07, FGJK08]. For instance, when it comes to the problem for deciding whether a set of CFDs can be satisfied by a nonempty database, the problem is NP-complete if attributes in the CFDs may have a finite domain, but it becomes tractable when all the attributes in the CFDs have an infinite domain [FGJK08]. In contrast, below we show that the consistency problem for editing rules is intractable even when all the attributes involved have an infinite domain.

Theorem 3.4.1:*[FLM$^+$12] The consistency problem for editing rules is coNP-complete, even when data and master relations have infinite-domain attributes only.* □

Theorem 3.4.1 tells us that the consistency analysis of eRs is more intricate than its CFD counterpart, which is in PTIME when all attributes involved have an infinite domain. It is also much harder than MDs, since any set of MDs is consistent [FGJ$^+$11]. Nevertheless, it is still decidable, as opposed to the undecidability for reasoning about rules for active databases [WC96].

The coverage problem. The *coverage problem* is to decide, given any (Z, T_c) and (Σ, D_m), whether (Z, T_c) is a certain region for (Σ, D_m). That is, whether (Σ, D_m) is able to fix errors in all the attributes of input tuples that are marked by (Z, T_c).

The coverage problem is, however, also intractable.

Theorem 3.4.2:*[FLM+12] The coverage problem is coNP-complete, even for input tuples and master relations that have infinite-domain attributes only.* □

Remark. Like the consistency and the coverage problems we have seen earlier, for all the problems to be studied in the rest of the section, their complexity remains the same in the presence of finite-domain attributes and in their absence. Hence in the sequel, we shall simply refer to their complexity bounds without remarking the absence of finite-domain attributes.

Special cases. To better understand these problems, we further investigate the following five special cases.

(1) Fixed Σ. In this setting, the set Σ of eRs is fixed. Indeed, editing rules are often predefined in practice.

(2) Fixed D_m. In this case the master data D_m is fixed. In real-life master data is changed less frequently than (input) data relations.

(3) Positive T_c. This case assumes no pattern tuples in T_c contain $\bar{a}$, *i.e.,* in the absence of negations.

(4) Concrete T_c. This case requires that no pattern tuples in T_c contain wildcard '_' or $\bar{a}$, *i.e.,* they contain a's only. Note that a concrete T_c must be a positive T_c.

(5) Direct fixes. We consider in this setting that (a) for all eRs $\varphi = ((X, X_m) \rightarrow (B, B_m), t_p[X_p])$ in Σ, $X_p \subseteq X$, *i.e.,* the pattern attributes X_p are also required to find a match in D_m, and (b) each step of a fixing process employs (Z, T_c) without extending (Z, T_c), *i.e.,* $t_{i-1} \rightarrow_{((Z,T_c),\varphi_i,t_{m_i})} t_i$.

Among these, cases (1) and (2) assume that Σ and D_m are fixed, respectively; (3) and (4) restrict the form of patterns in T_c; and case (5) restricts the form of eRs and adopts a simpler semantics for fixing input tuples.

One might think that fixed master data or positive patterns would simplify the analysis of eRs. Unfortunately, these do not help, due to the corollary follows.

Corollary 3.4.3:*[FLM+12] The consistency problem and the coverage problem remain coNP-complete even for (1) fixed master data* D_m *and (2) a positive tableau* T_c.

□

In contrast, special cases (1) and (4) indeed make our lives easier, as verified below.

Theorem 3.4.4:*[FLM$^+$12] The consistency problem and the coverage problem are in* PTIME *for either (1) a fixed set Σ of eRs or (2) a concrete pattern tableau T_c.* □

Furthermore, special case (5) identified above also simplifies the consistency and coverage analyses.

Theorem 3.4.5:*[FLM$^+$12] The consistency problem and the coverage problem are in* PTIME *when direct fixes are considered.* □

3.4.2 The Complexity of Computing Certain Regions

We next study three fundamental problems in connection with computing certain regions, when regions are either partially given or not given at all.

To derive a certain region (Z,T_c) from (Σ,D_m), one wants to know whether a given list Z of attributes could make a certain region by finding a *nonempty* T_c.

The *Z-validating* problem is to decide, given $(\Sigma,\ D_m)$ and a list Z of distinct attributes, whether there exists a non-empty pattern tableau T_c such that (Z,T_c) is a certain region for (Σ,D_m).

Another question is to determine, if Z can make a certain region by finding a nonempty T_c, how large T_c is. Let (Z,T_c) be a certain region for $(\Sigma,\ D_m)$. For any pattern tuple $t_c \in T_c$, we require the following:

(1) $t_c[A] = \_$ for all attributes A not appearing in Σ;

(2) $t_c[A]$ is replaced with v (resp. $\bar{v}$) if $t_c[A] = c$ (resp. $\bar{c}$) and c is a constant *not* appearing in Σ or D_m. Here v is a variable denoting any constant not in Σ or D_m.

Note that these requirements do not lose generality. It is easy to verify for any certain region (Z,T_c), we can find an equivalent one (with no more pattern tuples) satisfying the two conditions. Moreover, these allow us to deal with only a finite number of pattern tuples, and to focus on the essential properties of the problems.

The *Z-counting* problem is to count, given $(\Sigma,\ D_m)$ and a list Z of distinct attributes, the number of distinct pattern tuples that can be found from (Σ,D_m) to build a tableau T_c such that (Z,T_c) is a certain region.

Both problems are beyond reach in practice, as shown below. In particular, the Z-counting problem is as hard as finding the number of truth assignments that satisfy a given 3SAT instance [Pap94].

Theorem 3.4.6:*[FLM+12] The Z-validating problem is* NP*-complete.* □

In particular, in contrast to Theorem 3.4.5, the Z-validating problem remains intractable even when direct fixes are considered.

Corollary 3.4.7:*[FLM+12] The Z-validating problem remains* NP*-complete even when we consider (1) fixed master data D_m, (2) a positive pattern tableau T_c, (3) a concrete pattern tableau T_c, or (4) direct fixes.* □

However, when fixing Σ, the Z-validating problem becomes much simpler, as shown below.

Proposition 3.4.8:*[FLM+12] The Z-validating problem is in* PTIME *given a fixed set Σ of eRs.* □

We next investigate the Z-counting problem.

Theorem 3.4.9:*[FLM+12] The Z-counting problem is* #P*-complete.* □

From Theorems 3.4.6, 3.4.9 and Corollary 3.4.7 it follows:

Corollary 3.4.10:*[FLM+12] The Z-counting problem remains* #P*-complete even when we consider (1) fixed master data D_m, (2) a positive pattern tableau T_c, (3) a concrete pattern tableau T_c, or (4) direct fixes.* □

When only a fixed set Σ of eRs is considered, the Z-counting problem becomes easier. This is consistent with Proposition 3.4.8.

Proposition 3.4.11:*[FLM+12] The Z-counting problem is in* PTIME *given a fixed set Σ of eRs.* □

Certain regions with minimum Z. One would naturally want a certain region (Z, T_c) with a "small" Z, such that the users only need to assure the correctness of a small number of attributes in input tuples.

The *Z-minimum* problem is to decide, given (Σ, D_m) and a positive integer K, whether there exists a list Z of distinct attributes such that (a) $|Z| \leq K$ and (b) there exists a *non-empty* pattern tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m).

This problem is also intractable, as shown below.

Theorem 3.4.12:*[FLM+12] The Z-minimum problem is* NP*-complete.* □

The next corollary follows.

Corollary 3.4.13:*[FLM+12] The Z-minimum problem remains* NP*-complete even when we consider (1) fixed master data D_m, (2) a positive pattern tableau T_c, or (3) a concrete pattern tableau T_c.* □

When direct fixes are considered, the Z-minimum problem remains intractable, as opposed to Theorem 3.4.5.

Theorem 3.4.14:*[FLM$^+$12] The Z-minimum problem remains NP-complete even when direct fixes are considered.* □

The problem is in NP by Theorem 3.4.12.

Having seen Propositions 3.4.8 and 3.4.11, it is not surprising to find that the Z-minimum problem becomes tractable for a fixed set Σ of eRs, as shown below.

Proposition 3.4.15:*[FLM$^+$12] The Z-minimum problem is in* PTIME *given a fixed set* Σ *of eRs.* □

Problems	General setting/Infinite-domain attributes only					
	General	**Fixed Σ**	**Fixed D_m**	**Positive T_c**	**Concrete T_c**	**Direct Fixes**
Consistency	coNP-complete (Theorem 3.4.1)	PTIME (Theorem 3.4.4)	coNP-complete (Corollary 3.4.3)	coNP-complete (Corollary 3.4.3)	PTIME (Theorem 3.4.4)	PTIME (Theorem 3.4.5)
Coverage	coNP-complete (Theorem 3.4.2)	PTIME (Theorem 3.4.4)	coNP-complete (Corollary 3.4.3)	coNP-complete (Corollary 3.4.3)	PTIME (Theorem 3.4.4)	PTIME (Theorem 3.4.5)
Z-validating	NP-complete (Theorem 3.4.6)	PTIME (Proposition 3.4.8)	NP-complete (Corollary 3.4.7)	NP-complete (Corollary 3.4.7)	NP-complete (Corollary 3.4.7)	NP-complete (Corollary 3.4.7)
Z-counting	#P-complete (Theorem 3.4.9)	PTIME (Proposition 3.4.11)	#P-complete (Corollary 3.4.10)	#P-complete (Corollary 3.4.10)	#P-complete (Corollary 3.4.10)	#P-complete (Corollary 3.4.10)
Z-minimum	NP-complete (Theorem 3.4.12)	PTIME (Proposition 3.4.15)	NP-complete (Corollary 3.4.13)	NP-complete (Corollary 3.4.13)	NP-complete (Corollary 3.4.13)	NP-complete (Theorem 3.4.14)
	non-approx* (Theorem 3.4.17)	PTIME (Proposition 3.4.15)	non-approx (Corollary 3.4.18)	non-approx (Corollary 3.4.18)	non-approx (Corollary 3.4.18)	non-approx (Theorem 3.4.19)

non-approx: cannot be approximated within $c\log n$ in PTIME for a constant c, unless P = NP

Table 3.3: Summary of complexity results

Approximation hardness. Worse still, there exist no approximate algorithms for the (optimization version) Z-minimum problem with a reasonable bound. To show the approximation bound, we adopt L-reductions [Pap94].

Let Π_1 and Π_2 be two minimization problems. An L-reduction from Π_1 to Π_2 is a quadruple (f, g, α, β), where f and g are two PTIME computable functions, and α and β are two constants, such that

- ○ for any instance I_1 of Π_1, $I_2 = f(I_1)$ is an instance of Π_2 such that $\mathsf{opt}_2(I_2) \leq \alpha \cdot \mathsf{opt}_1(I_1)$, where opt_1 (resp. opt_2) is the objective of an optimal solution to I_1 (resp. I_2), and
- ○ for any solution s_2 to I_2, $s_1 = g(s_2)$ is a solution to I_1 such that $\mathsf{obj}_1(s_1) \leq \beta \cdot \mathsf{obj}_2(s_2)$, where $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) is a function measuring the objective of a solution to I_1 (resp. I_2).

We say an algorithm $\mathcal{A}$ for a minimization problem has performance guarantee ε ($\varepsilon \geq 1$) if for any instance I, $\mathsf{obj}(\mathcal{A}(I)) \leq \varepsilon \cdot \mathsf{opt}(I)$.

L-reductions retain approximation bounds [Pap94].

Proposition 3.4.16:*[FLM*$^+$*12] If* (f, g, α, β) *is an* L-reduction *from problems* Π_1 *to* Π_2*, and there is a* PTIME *algorithm for* Π_2 *with performance guarantee* ε*, then there is a* PTIME *algorithm for* Π_1 *with performance guarantee* $\alpha\beta\varepsilon$ *[Pap94].* □

Leveraging Proposition 3.4.16, we next show the approximation-hardness of the Z-minimum problem.

Theorem 3.4.17:*[FLM*$^+$*12] Unless NP* = P *, the Z-minimum problem cannot be approximated within a factor of* $c \log n$ *in* PTIME *for a constant c.* □

From Theorem 3.4.17 and Corollary 3.4.13, the result below immediately follows.

Corollary 3.4.18:*[FLM*$^+$*12] Unless NP* = P *, the Z-minimum problem cannot be approximated within a factor of* $c \log n$ *in* PTIME *for a constant c even when we consider (1) a fixed master relation* D_m*, (2) a positive pattern tableau* T_c*, or (3) a concrete pattern tableau* T_c*.* □

Direct fixes do not make our lives easier when approximation is concerned either, similar to Theorem 3.4.14.

Theorem 3.4.19:*[FLM*$^+$*12] Unless NP* = P *, the Z-minimum problem cannot be approximated within a factor of* $c \log n$ *in* PTIME *for a constant c for direct fixes.* □

Theorems 3.4.17, 3.4.19 and Corollary 3.4.18 tell us that to find certain regions, it is necessary to develop heuristic algorithms. Such algorithms are provided in [FLM$^+$10].

Summary. The complexity results are summarized in Table 3.3. Observe the following.

(1) The complexity bounds of all these problems remain unchanged in the presence of finite-domain attributes and in the absence of such attributes, as opposed to the analyses of CFDs [FGJK08], CINDs [BFM07] and MDs [FGJ$^+$11].

(2) For a fixed set Σ of eRs, all the problems become PTIME computable, *i.e.,* fixed eRs simplify the analyses.

(3) For fixed master data D_m or a positive tableau T_c, all the problems remain intractable. That is, these special cases do not make our lives easier.

(4) When we consider direct fixes or a concrete tableau T_c, the consistency problem and the coverage problem become tractable, while the other problems remain intractable. That is, these special cases simplify the analyses, but only to an extent. Due to the space constraint, we encourage the interested reader to consult [FLM$^+$10] for algorithms and experimental results based on direct fixes, which illustrate the practical impact of direct fixes.

3.5 An Interactive Framework for Certain Fixes

We next present a framework to find certain fixes for tuples at the point of data entry, by making use of editing rules and master data, and by interacting with users.

As depicted in Fig. 3.2, the framework is provided with a master relation D_m of schema R_m and a set Σ of eRs defined on (R, R_m). It takes a tuple t of schema R as input, and warrants to find a certain fix for t.

The algorithm underlying the framework, referred to as CertainFix, is shown in Fig. 3.3. The algorithm interacts with users and finds a certain fix for t as follows.

(1) Initialization (lines 1-2). It first picks a precomputed certain region (Z, T_c), and recommends Z as the first suggestion to the users (line 1). For an input tuple t, if $t[Z]$ is assured correct and if $t[Z]$ matches a pattern tuple in T_c, then a certain fix can be found for t. It also uses a set Z' to keep track of the attributes of t that are already fixed, which is initially empty (line 2).

As shown by Theorems 3.4.12 and 3.4.17, it is intractable and approximation-hard

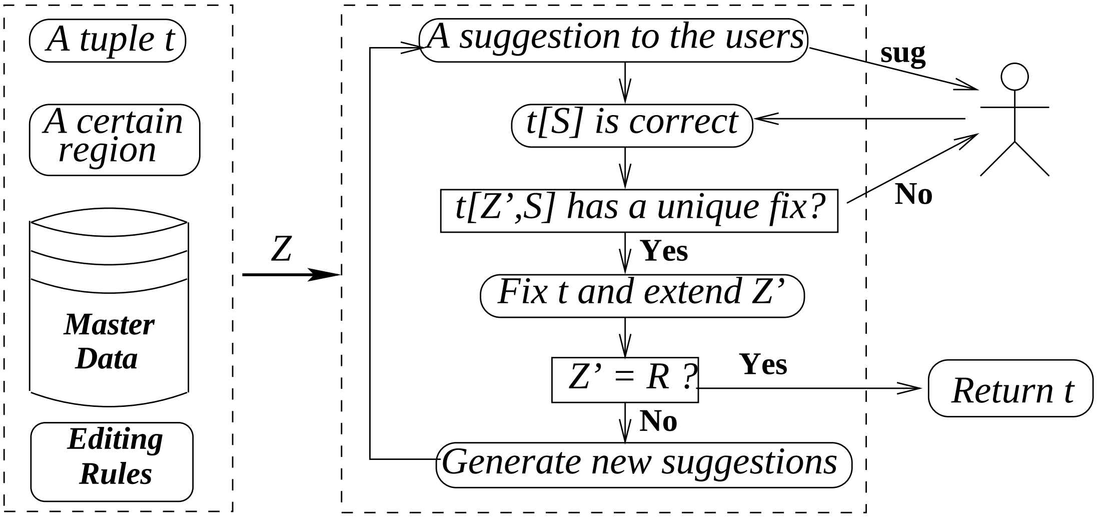


Figure 3.2: Framework overview

to find a certain region with a minimum set Z of attributes. Nevertheless, an efficient heuristic algorithm is provided by [FLM$^+$10], which is able to derive a set of certain regions from Σ and D_m based on a quality metric. Algorithm CertainFix picks the precomputed region (Z, T_c) with the highest quality. The region is computed once and is repeatedly used as long as Σ and D_m are unchanged.

(2) Generating correct fixes (lines 3-7). In each round of interaction with users, a set sug of attributes is recommended to the users as a suggestion (line 4), initially Z. The users get back with a set S of attributes that are asserted correct (line 5), where S may *not* necessarily be the same as sug. The algorithm validates $t[S]$ by checking whether $t[Z' \cup S]$ leads to a unique fix, *i.e.,* whether $t[S]$ is indeed correct. If $t[S]$ is invalid, the users are requested to revise the set S of attributes assured correct (line 6). If $t[Z' \cup S]$ yields a unique fix, procedure TransFix is invoked to find the fix, which extends Z' by including the newly corrected attributes (line 7). it finds the unique fix by invoking a procedure TransFix.

(3) Generating new suggestions (lines 8-9). If at this point, Z' covers all the attributes of R, the entire tuple t is validated and the fixed t is returned (lines 8, 10). Otherwise it computes a new suggestion from Σ and D_m via procedure Suggest (line 9), which is recommended to the users in the next round of interaction.

This process proceeds until a certain fix is found for t. All the attributes of t are corrected or validated, by using the users' input, the eRs and the master data.

The framework aims to guarantee the following. (a) *The correctness.* Each correct-

```
Input:   A tuple $t$, a certain region $(Z, T_c)$,
         a set $\Sigma$ of eRs, and a master relation $D_m$.
Output:  A fixed tuple $t$.

1.  sug := $Z$; /* $Z$ is the initial suggestion */
2.  $Z'$ := $\emptyset$;   flag := true;
3.  while flag do
4.    recommend sug to the users;
5.    input $S$, where $t[S]$ is assured correct and $S \cap Z' = \emptyset$;
6.    if $t[Z' \cup S]$ does not yield a unique fix then
        request new input from the users (back to line 4);
7.    $(t, Z')$ := TransFix ($t$, $Z' \cup S$, $D_m$, $\mathbf{G}$);
8.    if $Z' = R$ then flag := false;
9.    sug:= Suggest ($t$, $Z'$, $\Sigma$, $D_m$);
10. return $t$.
```

Figure 3.3: Algorithm CertainFix

ing step is justified by using the eRs and the master data. (b) *Minimizing user efforts*. It requires the users to validate a minimal number of attributes, while automatically deducing other attributes that are entailed correct. (c) *Minimal delays*. It improves the response time by reducing the latency for generating new suggestions at each interactive step.

Note that the users are not necessarily domain experts, as long as they can assure the correctness of certain attributes of input tuples that are required to match eRs and master tuples. In practice, different people may be responsible for entering and interpreting different attributes. Hence distinct attributes are often inspected and validated by different people.

In the rest of the section we present the details of the procedures and optimization techniques employed by CertainFix. Note that it is in PTIME to check whether $t[Z' \cup S]$ leads to a unique fix. Therefore, below we focus on TransFix and Suggest.

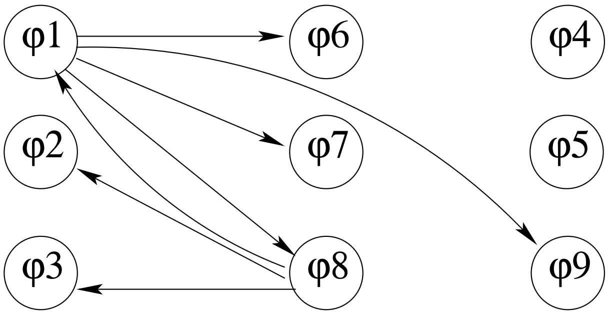


Figure 3.4: An example dependency graph

3.5.1 TransFix: Generating Correct Fixes

We first present procedure TransFix. It takes as input a tuple t, a master relation D_m, a set Σ of eRs, a set Z' of attributes such that $t[Z']$ has been validated. It finds a unique fix for t and extends Z' by including those newly validated attributes. While not all of the attributes of t may be validated, the procedure ensures that the attributes updated are correct.

Procedure TransFix represents Σ as a dependency graph $\mathbf{G}$, which tells us the order of applying eRs.

Dependency graph. The *dependency graph* $\mathbf{G}$ of a set Σ of eRs is a directed graph (V,E). Each node $v \in V$ denotes an eR $\varphi_v = ((X_v, X_{m_v}) \rightarrow (B_v, B_{m_v}), t_{p_v}[X_{p_v}])$. There exists an edge $(u,v) \in E$ from node u to v if $B_u \cap (X_v \cup X_{p_v}) \neq \emptyset$. Intuitively, (u,v) indicates that whether φ_v can be applied to t depends on the outcome of applying φ_u to t. Hence φ_u is applied before φ_v.

The dependency graph of Σ remains unchanged as long as Σ is not changed. Hence it is computed once, and is used to repair all input tuples until Σ is updated.

Example 3.5.1: The set Σ_0 of eRs given in Example 3.2.1 consists of 9 eRs, fully expressed as follows:

φ_1: $((\mathsf{zip}, \mathsf{zip}) \rightarrow (\mathsf{AC}, \mathsf{AC}), t_{p1} = ())$;
φ_2: $((\mathsf{zip}, \mathsf{zip}) \rightarrow (\mathsf{str}, \mathsf{str}), t_{p2} = ())$;
φ_3: $((\mathsf{zip}, \mathsf{zip}) \rightarrow (\mathsf{city}, \mathsf{city}), t_{p3} = ())$;
φ_4: $((\mathsf{phn}, \mathsf{Mphn}) \rightarrow (\mathsf{FN}, \mathsf{FN}), t_{p4}[\mathsf{type}] = (2))$;
φ_5: $((\mathsf{phn}, \mathsf{Mphn}) \rightarrow (\mathsf{LN}, \mathsf{LN}), t_{p5}[\mathsf{type}] = (2))$;
φ_6: $(([\mathsf{AC}, \mathsf{phn}], [\mathsf{AC}, \mathsf{Hphn}]) \rightarrow (\mathsf{str}, \mathsf{str}), t_{p6}[\mathsf{type}, \mathsf{AC}] = (1, \overline{0800}))$;
φ_7: $(([\mathsf{AC}, \mathsf{phn}], [\mathsf{AC}, \mathsf{Hphn}]) \rightarrow (\mathsf{city}, \mathsf{city}), t_{p7}[\mathsf{type}, \mathsf{AC}] = (1, \overline{0800}))$;
φ_8: $(([\mathsf{AC}, \mathsf{phn}], [\mathsf{AC}, \mathsf{Hphn}]) \rightarrow (\mathsf{zip}, \mathsf{zip}), t_{p8}[\mathsf{type}, \mathsf{AC}] = (1, \overline{0800}))$;

φ_9: $((\mathsf{AC}, \mathsf{AC}) \to (\mathsf{city}, \mathsf{city}), t_{p9}[\mathsf{AC}] = (\overline{0800}))$.

The dependency graph of Σ_0 is depicted in Fig. 3.4. Note that, for instance, there is an edge from φ_1 to φ_6 since the RHS of φ_1 (*i.e.,* $\{\mathsf{AC}\}$) is the subset of LHS of φ_6 (*i.e.,* $\{\mathsf{AC}, \mathsf{phn}\}$); similarly for the other edges. □ □

Procedure. Procedure TransFix is given in Fig. 3.5. It validates attributes of t as follows. It first marks all the nodes in the dependency graph as *unusable* (line 1). It then collects those nodes (eRs) whose LHS and pattern attributes are validated, puts them in a set vset (line 2), and marks them as *usable* (line 3). Intuitively, for the eR φ_v represented by a usable v, the attributes in $t[X_v \cup X_{p_v}]$ have already been validated, and hence, φ_v can be possibly applied to t. The procedure uses another set uset to maintain those eRs that are not yet usable but may become usable later on (line 4).

The procedure iteratively makes use of eRs in vset to fix attributes of t, and upgrades eRs from uset to vset (lines 5-15). In each iteration, a node v is randomly picked and removed from vset (line 6). If a master tuple t_m can be found such that (t_m, φ_v) applies to t, and moreover, if for the RHS attribute B_v of φ_v, $t[B_v]$ is not yet validated (line 7), then $t[B_v]$ is fixed using φ_v and t_m, and B_v is included in Z' (line 8).

The procedure then inspects each edge (v,u) emanating from v, to examine whether φ_u becomes usable (lines 9-15). If u is in the candidate set uset, and moreover, if RHS(φ_u) and RHS$_p$(φ_u) are included in the extended Z' (line 10), then u is added to vset, removed from uset (line 11), and is marked usable (line 12). Otherwise, if u is in neither vset nor uset (line 13), node u is added to vset if $X_u \cup X_{p_u}$ is a singleton set containing B_v (line 14), or to uset if $X_u \cup X_{p_u}$ contains other attributes besides B_v (line 15). Finally, the tuple t is returned along with the extended Z' (line 16).

Example 3.5.2: Consider tuple t_1 and the master data D_m of Fig. 3.1, and the set Σ_0 of eRs given in Example 3.5.1. Assume that Z consists of zip only. Given D_m, Z and the dependency graph **G** of Fig, 3.4, we show how procedure TransFix fixes attributes of t_1. As indicated in the table below, in iteration 0, uset is empty, while φ_1 is in vset since its $X \cup X_p \subseteq Z'$; similarly for φ_2 and φ_3.

iteration	Z'	vset	uset
0	zip	φ_1, φ_2, φ_3	$\emptyset$
1	zip, AC	φ_2, φ_3, φ_9	φ_6, φ_7, φ_8
2	zip, AC, str	φ_3, φ_9	φ_6, φ_7, φ_8
3	zip, AC, str, city	φ_9	φ_6, φ_7, φ_8
4	zip, AC, str, city	$\emptyset$	φ_6, φ_7, φ_8

```
Input: A tuple $t$, a set $Z'$ of attributes, a master relation $D_m$,
       and a dependency graph $\mathbf{G}(V,E)$.
Output: A (partially) fixed tuple $t$ and validated attributes $Z'$.

/* node $u$: $\varphi_u = ((X_u, X_{m_u}) \rightarrow (B_u, B_{m_u}), t_{p_u}[X_{p_u}])$ */;
mark $u$.usable := false for each $u \in V$;
vset $:= \{ u \mid u \in V$ and $(X_u \cup X_{p_u}) \subseteq Z' \}$;
mark $u$.usable := true for each $u \in$ vset;
uset $:= \{ u \mid u \in V$ and $(X_u \cup X_{p_u}) \nsubseteq Z'$
             and $(X_u \cup X_{p_u}) \cap Z' \neq \emptyset \}$;
while vset $\neq \emptyset$ do
    $v$ := an eR picked from vset;  vset := vset $\setminus \{v\}$;
    if $\exists t_m \in D_m$, $(t_m, \varphi_v)$ applies to $t$ and $B_v \notin Z'$ then
        $t[B_v] := t_m[B_{m_v}]$;  $Z' := Z' \cup \{B_v\}$;
        for each edge $(v,u) \in E$ do
            if $u \in$ uset  and  $(X_u \cup X_{p_u}) \subseteq Z'$ then
                vset := vset $\cup \{u\}$;  uset := uset $\setminus \{u\}$;
                $u$.usable := true;
            else if $u \notin$ uset  and  $u$.usable = false then
                if $\{B_v\} = (X_u \cup X_{p_u})$ then vset := vset $\cup \{u\}$;
                else if $B_v \in (X_u \cup X_{p_u})$ then uset := uset $\cup \{u\}$;
return $(t, Z')$;
```

Figure 3.5: Procedure TransFix

In iteration 1, TransFix picks and removes φ_1 from vset. It finds that φ_1 and master tuple s_1 (in Fig. 3.1) can be applied to t_1. Hence it normalizes $t_1[\text{AC}] := s_1[\text{AC}] = 131$, and expands Z' by including AC. It adds φ_9 to vset since $X \cup X_p$ of φ_9, *i.e.,* $\{\text{AC}\}$, is validated. Moreover, φ_6–φ_8 are added to uset, since while AC is validated, attributes phn and type are not yet.

In iteration 2 (resp. 3), φ_2 (resp. φ_3) is selected from vset, and str (resp. city) is fixed by matching s_1. Here t_1 is updated by $t_1[\text{str}] := s_1[\text{str}] = 51$ Elm Row.

In iteration 4, φ_9 is selected and removed from vset. No change is incurred to t since city is already validated. TransFix terminates since vset is now empty. □ □

Correctness. Observe the following. (1) Each eR is used at most once. When a node is removed from vset, it will not be put back. Since the size of vset is at most the number card(Σ) of eRs in Σ, the while loop (lines 5-15) iterates at most card(Σ) times. (2) When applying (t_m, φ) to t, $t[X \cup X_p]$ have already been validated; thus $t[B]$ is ensured correct. (3) All the eRs that are possibly usable are examined. Hence, when TransFix terminates, no more attributes of t could be fixed given Z.

Complexity. Let $\mathbf{G}(V,E)$ be the dependency graph of Σ. Note that $|V| = \mathsf{card}(\Sigma)$. The initialization of TransFix runs in $O(|\Sigma|)$ time (lines 1-4), by employing a hash table. As argued above, at most $|V|$ iterations of the outer loop (lines 6-15) are executed, since each iteration consumes at least one eR in Σ. The inner loop (lines 10-15) is run at most $|V|$ times for each outer iteration (*i.e.,* checking all eRs in Σ). In addition, observe the following: (a) checking containment and intersection of two attribute sets $(X_u \cup X_{p_u})$ and Z' is in $O(|X_u \cup X_{p_u}|)$ time if we use a hash table; and (b) it takes constant time to check whether there exists a master tuple that is applicable to t with an eR, by using a hash table that stores $t_m[X_m]$ as a key for $t_m \in D_m$. Putting these together, each outer iteration is in $O(|\Sigma|)$ time, and hence, TransFix is in $O(|V||\Sigma|)$ time, which is at most $O(|\Sigma|^2)$. In practice, $|\Sigma|$ is typically small.

3.5.2 Suggest: Generating New Suggestions

To present procedure Suggest, we first define *suggestions* and state the problem of finding suggestions.

Suggestions. Consider a tuple t, where $t[Z]$ has been validated. A *suggestion* for t *w.r.t.* $t[Z]$ is a set S of attributes such that there exists a certain region $(Z \cup S, \{t_c\})$, where t_c is a pattern and $t[Z]$ satisfies $t_c[Z]$.

That is, if the users additionally assert that $t[S]$ is correct and $t[Z \cup S]$ matches some certain region, then a certain fix is warranted for t.

Example 3.5.3: Recall from Example 3.5.2 that $t_1[Z]$ is fixed by using Σ_0 and D_m, where $Z = \{\mathsf{zip}, \mathsf{AC}, \mathsf{str}, \mathsf{city}\}$. Let $S = \{\mathsf{phn}, \mathsf{type}, \mathsf{item}\}$. One can verify that S is a suggestion for t_1 *w.r.t.* $t_1[Z]$. Indeed, $(Z \cup S,\ \{t_c\})$ is a certain region for (Σ_0, D_m), where $t_c = (\underbrace{\text{EH7 4AH}, 131, \text{51 Elm Row}, \text{Edi}}_{Z}, \underbrace{079172485, 2, \_}_{S})$. □ □

The users would naturally want a suggestion as "small" as possible, so that they need to make minimal efforts to ensure some attributes of t to be correct. This motivates us to study the following problem.

The *S-minimum problem* is to decide, given (Σ, D_m), a set $t[Z]$ of attributes that has been validated, and a positive integer K, whether there exists a non-empty set S of attributes such that (a) $Z \cap S = \emptyset$, (b) $|S| \leq K$ and (c) S is a suggestion for t *w.r.t.* $t[Z]$.

Observe that the Z-minimum problem (Section 3.4) is a special case of the S-minimum problem when no attribute is fixed initially (*i.e.,* $Z = \emptyset$). From this and Theorems 3.4.12 and 3.4.17 it follows that the S-minimum problem is NP-complete and approximation-hard.

These complexity bounds suggest that we develop heuristic algorithms to compute suggestions, along the same lines as computing certain regions, as discussed in [FLM$^+$10]. When computing Z-minimum certain regions, all eRs need to be considered [FLM$^+$10]. When it comes to suggestions, in contrast, attributes $t[Z]$ are already validated, which can be used to reduce the search space of eRs by refining some eRs and leaving the others out.

To do this we use the following notations. For an eR $\varphi = ((X, X_m) \rightarrow (B, B_m), t_p[X_p])$ and a list X_i of attributes in X, we use $\lambda_\varphi(X_i)$ to denote the corresponding attributes in X_m. For instance, when $(X_i, X_{m_i}) = (ABC, A_mB_mC_m)$, $\lambda_\varphi(AC) = A_mC_m$. We also write $\varphi^+ = ((X, X_m) \rightarrow (B, B_m), t_p^+[X_p^+])$, where $X_p \subseteq X_p^+$, *i.e.,* φ^+ differs from φ only in the pattern.

Consider a set Σ of eRs, a master relation D_m, an input tuple t, and attributes Z such that $t[Z]$ is fixed using TransFix. For an eR φ in Σ, (1) if there exists no tuple $t_m \in D_m$ such that (φ, t_m) applies to t, then φ cannot be used to fix t; otherwise, (2) we may extend the pattern of φ and refine its values with $t[Z]$, which yields φ^+. Hence we introduce the following notion.

The set of *applicable rules* for $t[Z]$ *w.r.t.* Σ, denoted as $\Sigma_{t[Z]}$, consists of eRs φ^+ defined as follows. For each φ in Σ, φ^+ is derived from φ if (a) $B \not\in Z$; (b) $t_p[X_p \cap Z] \approx t[X_p \cap Z]$; and (c) there exists a master tuple $t_m \in D_m$, where $t_m[\lambda_\varphi(X_p \cap X)] \approx t_p[X_p \cap X]$ and $t_m[\lambda_\varphi(X \cap Z)] = t[X \cap Z]$. Here in φ^+, (i) $X_p^+ = X_p \cup (X \cap Z)$ and (ii) $t_p^+[X_p^+ \cap Z] = t[X_p^+ \cap Z]$.

Intuitively, φ^+ can be derived from φ if φ does not change the validated attributes (*i.e.,* (a) above), matches them (*i.e.,* (b)), and moreover, if there exists some master tuple that can be applied to t with φ (*i.e.,* (c)). The refined rule φ^+ extends the pattern attributes of φ with Z (*i.e.,* (i) above), and enriches its pattern values using the specific values of $t[Z]$ (*i.e.,* (ii)).

Example 3.5.4: For t_1[zip, AC, str, city] validated in Example 3.5.2, applicable rules in $\Sigma_{t_1[\text{zip,AC,str,city}]}$ include:

φ_4: $((\mathsf{phn}, \mathsf{Mphn}) \to (\mathsf{FN}, \mathsf{FN}), t_{p4}[\mathsf{type}] = (2))$;
φ_5: $((\mathsf{phn}, \mathsf{Mphn}) \to (\mathsf{LN}, \mathsf{LN}), t_{p5}[\mathsf{type}] = (2))$;
φ_6^+: $(([\mathsf{AC}, \mathsf{phn}], [\mathsf{AC}, \mathsf{Hphn}]) \to (\mathsf{str}, \mathsf{str}), t_{p6}[\mathsf{type}, \mathsf{AC}]=(1,131))$;
φ_7^+: $(([\mathsf{AC}, \mathsf{phn}], [\mathsf{AC}, \mathsf{Hphn}]) \to (\mathsf{city}, \mathsf{city}), t_{p7}[\mathsf{type}, \mathsf{AC}]=(1,131))$;
φ_8^+: $(([\mathsf{AC}, \mathsf{phn}], [\mathsf{AC}, \mathsf{Hphn}]) \to (\mathsf{zip}, \mathsf{zip}), t_{p8}[\mathsf{type}, \mathsf{AC}]=(1,131))$;

Here φ_4 and φ_5 are taken from Σ_0, while φ_6^+ is derived from φ_6 by refining $t_{p6}[\mathsf{AC}]$ (from $\overline{0800}$ to 131), when $t_1[\mathsf{AC}]$ is known to be 131; similarly for φ_7^+ and φ_8^+. □ □

We show below that it suffices to consider $\Sigma_{t[Z]}$.

Proposition 3.5.1: *When $t[Z]$ is assured correct, S is a suggestion for t iff there exists a pattern tuple t_c such that $(Z \cup S, \{t_c\})$ is a certain region for $(\Sigma_{t[Z]}, D_m)$.* □

Proof. Assume that there exists t_c such that $(Z \cup S, \{t_c\})$ is a certain region for $(\Sigma_{t[Z]}, D_m)$. We show that S is a suggestion by constructing a pattern tuple t'_c such that $(Z \cup S, \{t'_c\})$ is a certain region for (Σ, D_m). Consider t'_c, where $t'_c[Z] = t[Z]$ and $t'_c[S] = t_c[S]$. One can easily verify the following. (1) $(Z \cup S, \{t'_c\})$ is a certain region for $(\Sigma_{t[Z]}, D_m)$; (2) the set of attributes covered by $(Z \cup S, \{t'_c\}, \Sigma, D_m)$ is the same as the set covered by $(Z \cup S, \{t'_c\}, \Sigma_{t[Z]}, D_m)$; and (3) $(\Sigma_{t[Z]}, D_m)$ is consistent *w.r.t.* $(Z \cup S, \{t'_c\})$ iff (Σ, D_m) is consistent *w.r.t.* $(Z \cup S, \{t'_c\})$. From these it follows that $(Z \cup S, \{t'_c\})$ is also a certain region for (Σ, D_m).

Conversely, assume that S is a suggestion. Then there exists a certain region $(Z \cup S, \{t_c\})$ for (Σ, D_m). We define a pattern tuple t'_c, where $t'_c[Z] = t[Z]$ and $t'_c[S] = t_c[S]$. One can show that $(Z \cup S, \{t'_c\})$ is a certain region for $(\Sigma_{t[Z]}, D_m)$. Indeed, this can be verified along the same lines as the argument given above. □

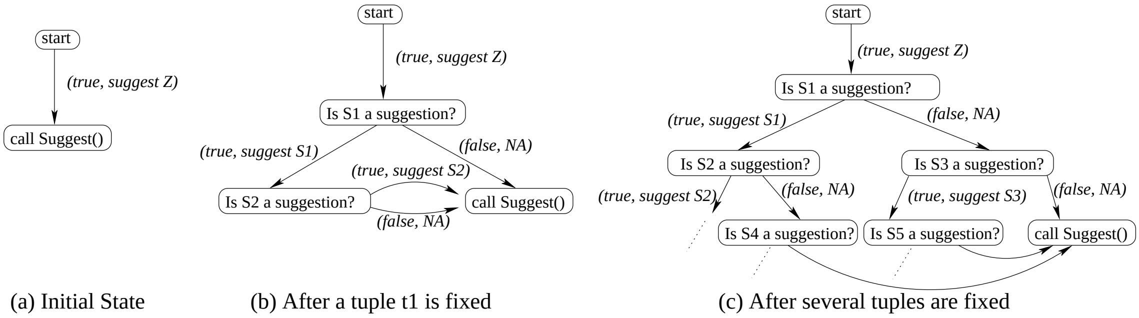
Procedure Suggest. Leveraging Proposition 3.5.1, we outline procedure Suggest in Fig. 3.6. It takes Σ, D_m, Z and t as input, and finds a suggestion as follows. It first derives applicable rules $\Sigma_{t[Z]}$ from Σ and $t[Z]$ (line 1). It then computes a certain region for $(\Sigma_{t[Z]}, D_m)$ (line 2), by employing the algorithm provided in [FLM$^+$10]. Finally, it constructs and returns a new suggestion (line 3).

Correctness and Complexity. The correctness of Suggest follows from the definition of suggestions and Proposition 3.5.1. For its complexity, observe the following. (1) The set $\Sigma_{t[Z]}$ can be derived from Σ and $t[Z]$ in $O(|\Sigma| + |t|)$ time, by employing the indices developed for Procedure TransFix. Indeed, the conditions for applicable rules can be checked in constant time. (2) The algorithm of [FLM$^+$10] computes a certain region in $O(|\Sigma_{t[Z]}|^2 |D_m| \log(|D_m|))$ time, where $|\Sigma_{t[Z]}| \leq |\Sigma|$. Hence Suggest is

```
Input: Tuple $t$, attributes $Z$, eRs $\Sigma$, and master data $D_m$.
Output: A set sug of attributes as suggestion.

1. derive $\Sigma_{t[Z]}$ using $t$, $Z$ and $\Sigma$;
2. compute a certain region $(Z', T_c)$ using $\Sigma_{t[Z]}$ and $D_m$;
3. return sug $:= Z' \setminus Z$;
```

Figure 3.6: Procedure Suggest



(a) Initial State (b) After a tuple t1 is fixed (c) After several tuples are fixed

Figure 3.7: A sample BDD

in $O(|\Sigma|^2|D_m|\log(|D_m|))$ time.

Optimization. It is quite costly to compute a certain region in each round of user interactions. This motivates us to develop an optimization strategy, which aims to minimize unnecessary recomputation by reusing certain regions computed earlier. In a nutshell, when processing a stream of input tuples of schema R, we maintain certain regions generated for them. When a new input tuple t arrives, we check whether some region computed previously remains a certain region for fixing t. If so, we simply reuse the region, without computing a new one starting from scratch. We compute new suggestions only when necessary. As will be verified by our experimental study, this reduces the cost significantly, since it is far less costly to check whether a region is certain than computing new certain regions [FLM$^+$10].

We maintain previously computed certain regions by using a *binary decision diagram* (BDD) [Knu09]. A BDD is a directed acyclic graph $\mathbf{G}_b = (V_b, E_b)$. Each node u in V_b represents either a condition or a call for Suggest, and it has at most two outgoing edges. The root of $\mathbf{G}_b$ is denoted as *start*. Each edge (u, v) is labeled with a pair

$(\mathsf{bval}, \mathsf{act})$, where bval is either Boolean value true or false; and act is an action, which provides a suggestion if bval is true, and generates new suggestions otherwise.

Example 3.5.5: Consider the evolution of a BDD depicted in Fig. 3.7. When no tuples have been processed, the BDD is shown in Fig. 3.7(a). Here the set Z of attributes taken from the precomputed certain region is treated as the first suggestion, as described in procedure TransFix. For the first input tuple t_1, if $t_1[Z]$ does not match any certain region, a new suggestion needs to be computed; hence the call for procedure Suggest.

Assume that t_1 is fixed with two suggestions S_1 and S_2. Then BDD is expanded, as shown in Fig. 3.7(b). Consider a newly arrived tuple t_2. If $t_2[Z]$ does not satisfy any certain region, TransFix expands the set Z' of validated attributes. We check whether S_1 is a suggestion *w.r.t.* $t_2[Z']$. If so, the true branch is followed and S_1 is recommended to the users; otherwise Suggest is invoked to generate a new suggestion. Similarly, S_2 is checked. If t_2 still cannot be fixed with S_2, Suggest is invoked for both the true and the false branches to produce a new suggestion. The new suggestion is added to the BDD.

After more tuples are fixed, the BDD may evolve to Fig. 3.7(c), which collects those certain regions generated when processing these tuples. As shown above, these regions are reused when processing new tuples. □ □

Capitalizing on BDD, we present an optimized Suggest, denoted as Suggest$^+$, which is outlined in Fig.3.8. It takes t, Z, Σ, D_m, a BDD G_b and a node u on G_b as input, and finds a suggestion as follows.

Suggest$^+$ traverses G_b top-down starting from its root, *i.e.,* the input u is initialized at *start* node. At each round of interaction, a node u of G_b is visited, at which it checks whether a precomputed suggestion associated with u remains a suggestion for t. If not, it checks other previously computed regions via a false branch (lines 1-2). Otherwise, it recommends the same suggestion to the users, and moves to the child of u via a true branch (lines 3-4). In the next round of interaction, if needed, checking resumes at node u. Suggest is invoked to compute new suggestions when no known regions can be reused, and G_b is also maintained (line 5). Finally, a suggestion is returned (line 6).

It implements a strategy to decide what suggestions are maintained by a BDD (line 5), to strike a balance between checking a set of suggestions and recomputing a certain region. It also compresses BDD to reduce the space cost. We omit the details for space limit.

We revise CertainFix by using Suggest$^+$ instead of Suggest, and refer to it as

```
Input: Tuple $t$, attributes $Z$, eRs $\Sigma$, master data $D_m$,
       a BDD $G_b(V_b, E_b)$, and a node $u$ in $V_b$.
Output: A set sug of attributes as suggestion.

while $u$ is not a suggestion and $u$ has a false branch do
    $u := v$ where the edge $(u, v)$ is a false branch;
if $u$ is a suggestion then
    get sug from $u$;    $u := v$ where $(u, v)$ is a true branch;
else sug := Suggest$(t, Z, \Sigma, D_m)$;    maintain $G_b$ with sug;
return sug;
```

Figure 3.8: Procedure Suggest$^+$

CertainFix$^+$.

3.6 CerFix: A System for Cleaning Data with Certain Fixes

Following [FLM$^+$10], we develop CERFIX, a data cleaning system that finds certain fixes for input tuples at the point of data entry. It differs from other systems that also ask for user feedback (*e.g.,* [CCC$^+$10]) in what feedback is requested and how the feedback is used. Below we first present CERFIX, and then outline what functionalities we shall demonstrate.

3.6.1 The CERFIX System

The architecture of CERFIX is depicted in Fig. 3.9. CERFIX maintains a collection of master data (master data manager) and a set of editing rules (rule engine). With respect to the master data and editing rules, it computes a set of certain regions (region finder). It inspects and repairs input tuples via a data monitor, which interacts with the users to find certain fixes for input tuples. It also provides a Web interface (data explorer) and a data auditing module for users to manage editing rules and trace changes to data, respectively.

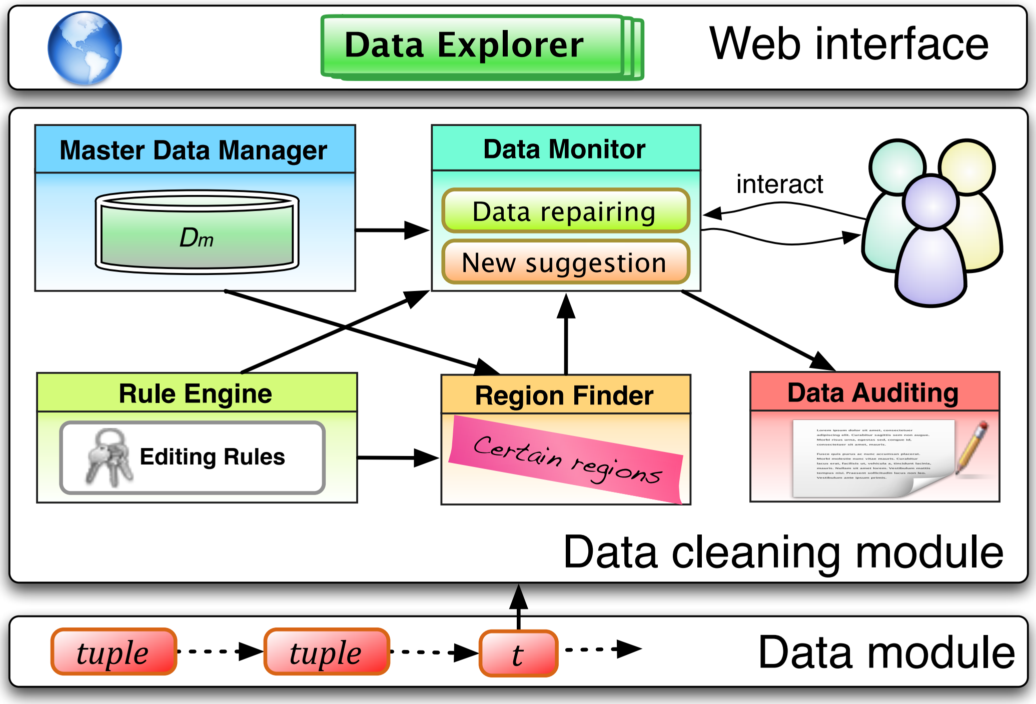


Figure 3.9: The CERFIX Architecture.

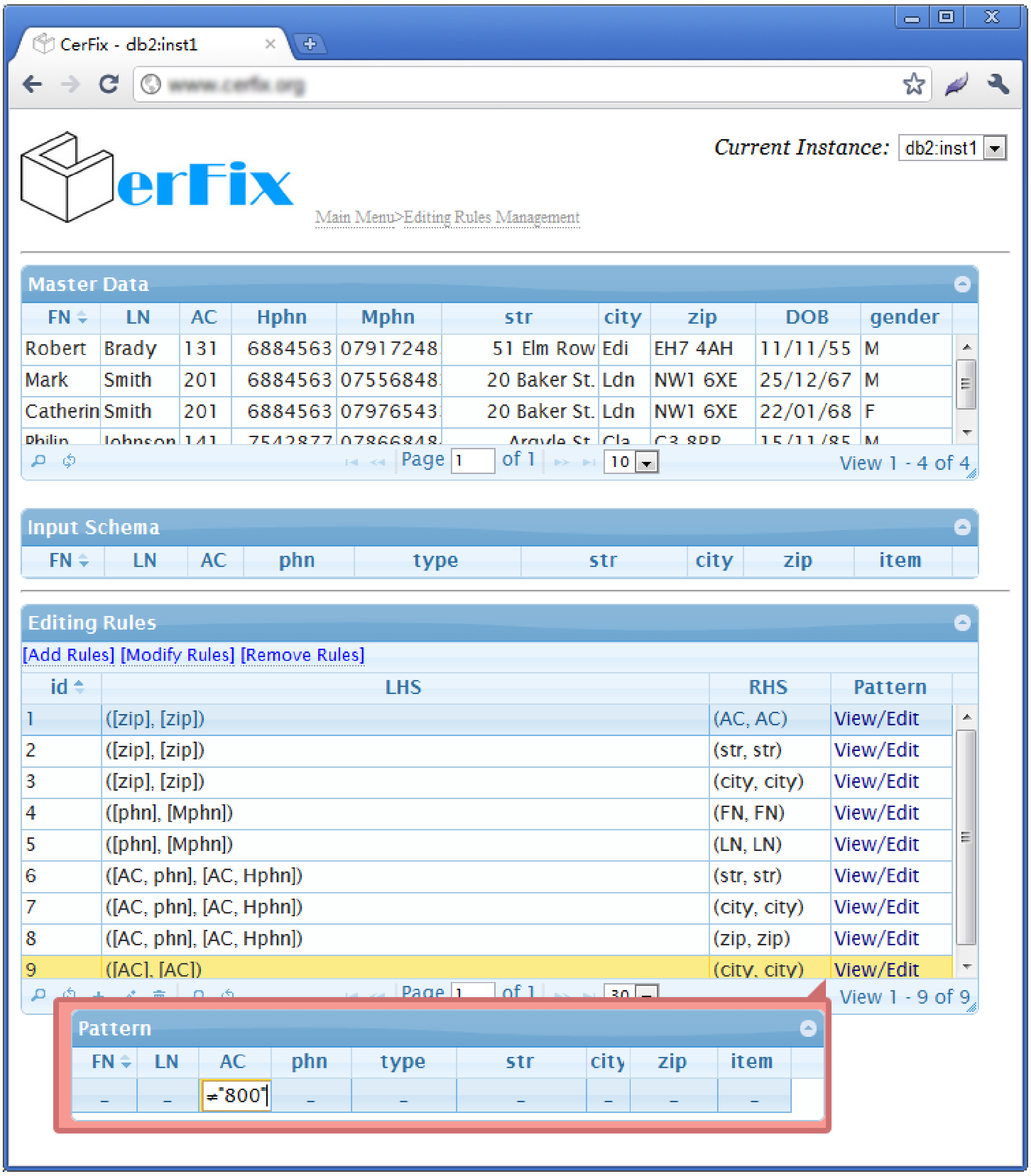
Below we briefly present the key components of CERFIX. We refer the interested readers to [FLM+10] for details about editing rules, certain regions and their computation.

Rule engine. It maintains a set of editing rules (eRs) that specify (a) whether an input tuple t can match a master tuple s via a pattern tuple, and (b) which attribute values of t can be changed and what correct values they should take from the master data. The engine also implements static analysis techniques developed in [FLM+10]. In particular, it supports the following. (1) It checks the consistency of editing rules, *i.e.,* whether the given rules are dirty themselves. Moreover, (2) provided that some attributes of a tuple are correct, it automatically derives what other attributes can be validated (assured correct) by using editing rules and master data.

Editing rules can be either explicitly specified by the users, or derived from integrity constraints, *e.g.,* CFDs and matching dependencies [FJLM09] for which discovery algorithms are already in place. CERFIX currently only supports manual specification of editing rules via the Web interface.

Master data manager. It maintains master data, which is assumed consistent and accurate [RW08].

Region finder. A region is a pair (Z, T_c), where Z is a list of attributes of an input tuple and T_c is a *pattern tableau* consisting of a set of pattern tuples with attributes in Z. A region (Z, T_c) is a *certain region w.r.t.* a set of editing rules and master data if for any input tuple t, as long as $t[Z]$ is correct and $t[Z]$ matches a pattern in T_c, the editing rules warrant to find a certain fix for t. Based on the algorithms in [FLM$^+$10], top-k certain regions are pre-computed that are ranked ascendingly by the number of attributes, and are recommended to users as (initial) suggestions.


CerFix - db2:inst1
CerFix
Current Instance: db2:inst1
Main Menu>Editing Rules Management

Master Data

FN	LN	AC	Hphn	Mphn	str	city	zip	DOB	gender
Robert	Brady	131	6884563	07917248	51 Elm Row	Edi	EH7 4AH	11/11/55	M
Mark	Smith	201	6884563	07556848	20 Baker St.	Ldn	NW1 6XE	25/12/67	M
Catherin	Smith	201	6884563	07976543	20 Baker St.	Ldn	NW1 6XE	22/01/68	F
Philip	Johnson	141	7542877	07866848	Argyle St	Gla	G3 8RR	15/11/85	M

Page 1 of 1 10 View 1 - 4 of 4

Input Schema

FN	LN	AC	phn	type	str	city	zip	item

Editing Rules

[Add Rules] [Modify Rules] [Remove Rules]

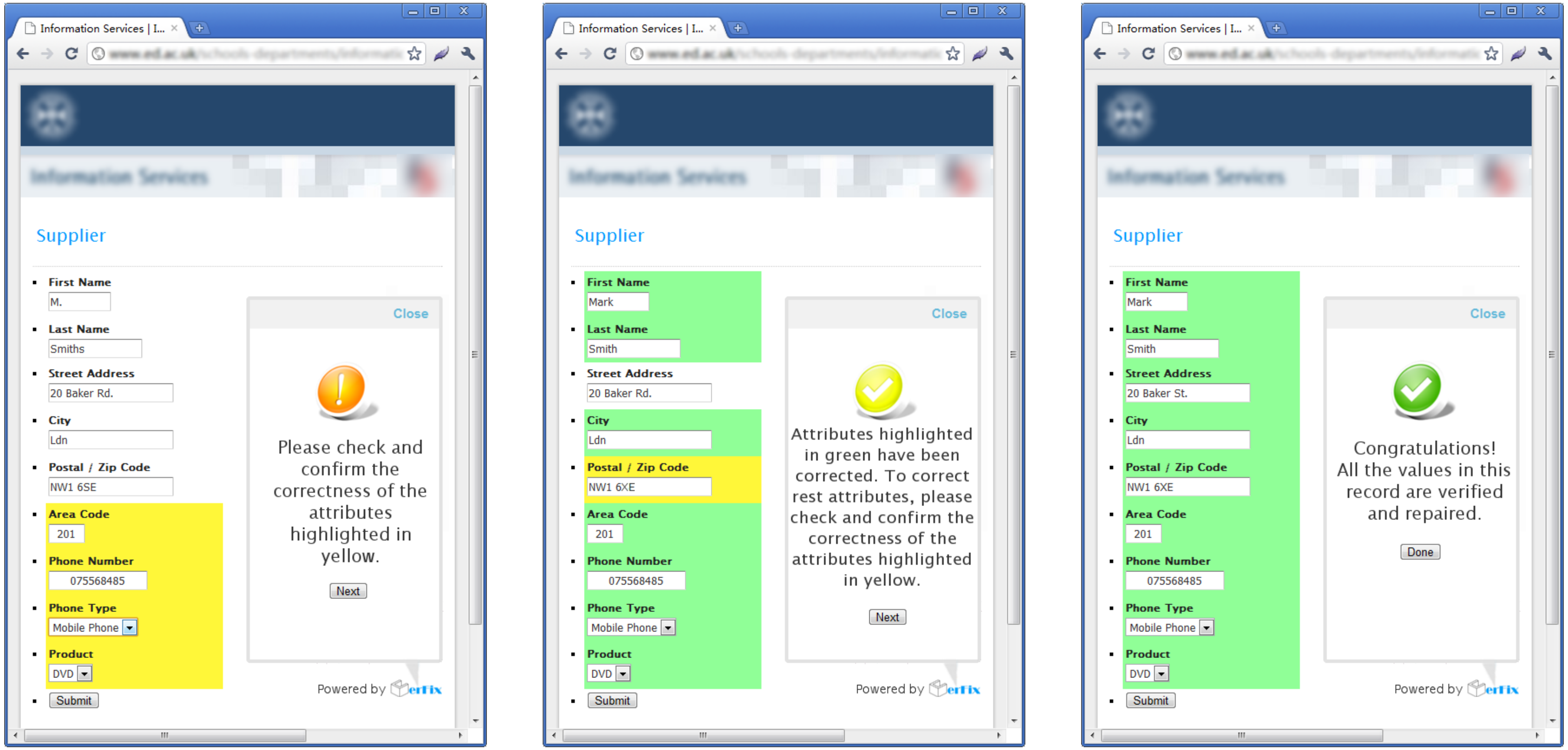
id	LHS	RHS	Pattern
1	([zip], [zip])	(AC, AC)	View/Edit
2	([zip], [zip])	(str, str)	View/Edit
3	([zip], [zip])	(city, city)	View/Edit
4	([phn], [Mphn])	(FN, FN)	View/Edit
5	([phn], [Mphn])	(LN, LN)	View/Edit
6	([AC, phn], [AC, Hphn])	(str, str)	View/Edit
7	([AC, phn], [AC, Hphn])	(city, city)	View/Edit
8	([AC, phn], [AC, Hphn])	(zip, zip)	View/Edit
9	([AC], [AC])	(city, city)	View/Edit

Page 1 of 1 30 View 1 - 9 of 9

Pattern

FN	LN	AC	phn	type	str	city	zip	item
_	_	≠"800"	_	_	_	_	_	_

Figure 3.10: Management of editing rules.



Information Services | I...

Supplier

- First Name: M.
- Last Name: Smiths
- Street Address: 20 Baker Rd.
- City: Ldn
- Postal / Zip Code: NW1 6SE
- Area Code: 201
- Phone Number: 075568485
- Phone Type: Mobile Phone
- Product: DVD
- Submit

Close

Please check and confirm the correctness of the attributes highlighted in yellow.

Next

Powered by cerFix

(a) Providing suggestions

Information Services | I...

Supplier

- First Name: Mark
- Last Name: Smith
- Street Address: 20 Baker Rd.
- City: Ldn
- Postal / Zip Code: NW1 6XE
- Area Code: 201
- Phone Number: 075568485
- Phone Type: Mobile Phone
- Product: DVD
- Submit

Close

Attributes highlighted in green have been corrected. To correct rest attributes, please check and confirm the correctness of the attributes highlighted in yellow.

Next

Powered by cerFix

(b) Fixing data

Information Services | I...

Supplier

- First Name: Mark
- Last Name: Smith
- Street Address: 20 Baker St.
- City: Ldn
- Postal / Zip Code: NW1 6XE
- Area Code: 201
- Phone Number: 075568485
- Phone Type: Mobile Phone
- Product: DVD
- Submit

Close

Congratulations! All the values in this record are verified and repaired.

Done

Powered by cerFix

(c) Achieving certain fixes

Figure 3.11: Data monitor.

Data monitor. This is the most important module of CERFIX. It interacts with the users and finds certain fixes for input tuples, as follows. (1) *Initial suggestions*. It recommends the set of certain regions computed by region finder to the users as suggestions. For each input tuple t, if the users ensure that $t[Z]$ is correct and matches a pattern in T_c for any region (Z, T_c) in the set, then a certain repair for t is warranted. (2) *Data repairing*. For an input tuple t, the users may respond with a set $t[S]$ of attributes that is correct, where S may *not* be any of the certain regions. Data monitor iteratively employs editing rules and master data to fix as many attributes in t as possible, and expands the correct attribute set S by including those attributes that are validated via the inference system of the rule engine. (3) *New suggestion*. If not all attributes of t have been validated, data monitor computes a new suggestion, *i.e.,* a *minimal* number of attributes, which are recommended to the users. If the users ensure the correctness of these attributes in t, data monitor will find a certain fix for t. The process of steps (2) and (3) repeats until a certain fix of t is reached.

CERFIX ensures that each fix is correct with editing rules and master data. It also minimizes users' effort by identifying a minimal number of attributes for users to validate.

Data auditing. This module keeps track of changes to each tuple, incurred either by the users or automatically by data monitor with editing rules and master data. Statistics about the changes can be retrieved upon users' requests.

3.6.2 How the CERFIX works

We next describe various aspects of CERFIX in more detail. More specifically, we show the following: (1) how users manage editing rules with the aid of the Web interface (data explorer in Fig. 3.9); (2) how CERFIX interacts with the users for data monitoring, to detect and fix errors in input tuples at the point of their entry; and (3) how data auditing works, to keep track of which attributes are fixed and where the correct values come from, and provide statistics about the percentage of data that is fixed by user efforts or by CERFIX.

Initialization. The users are required to configure an instance, which consists of two parts: (a) a data connection with JDBC url, username, password and the corresponding JDBC driver provided by users; and (b) specifying the schema of input (dirty) tuples and that of the master data.

We illustrate these with the master data and the input data shown in Fig. 3.10. Note that they have different schemas.

Master data. Each tuple in the master data specifies a person in the UK in terms of the name (FN, LN), area code (AC), home phone (Hphn), mobile phone (Mphn), address (street str, city and zip code), date of birth (DOB) and gender.

Input tuples. Each tuple specifies a UK customer.

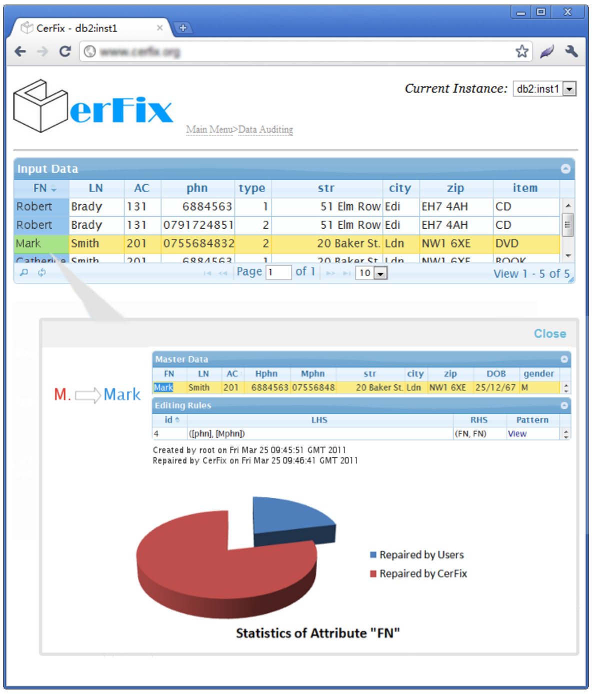
Editing rule management. Figure 3.10 displays the Web interface for managing eRs. We show how the users can manage (view/modify/add/delete) eRs using the data explorer. The system currently only supports to import eRs manually via the rule manager, where the eRs may either be designed by experts or be discovered from CFDs or MDs. For instance, Figure 3.10 shows nine editing rules φ_1–φ_9, for the id's 1–9, respectively.

- ◦ φ_1 (resp. φ_2 and φ_3) states that if an input tuple t and a master tuple s have the same zip code and if $t[\mathsf{zip}]$ is already validated, then t can be updated by $t[\mathsf{zip}] := s[\mathsf{zip}]$ (resp. str and city).
- ◦ φ_4 (resp. φ_5) states that if the phn of a tuple t matches the Mphn of a master tuple s, and if $t[\mathsf{phn}]$ is validated, then $t[\mathsf{FN}] := s[\mathsf{FN}]$ (resp. LN). These eRs pose a constraint (a pattern tuple) $t[\mathsf{type}] = 2$, requiring that phn is mobile phone. This rule can be viewed or edited by clicking the view/edit frame for the pattern.
- ◦ φ_6 (resp. φ_7 and φ_8) tells us that if the (AC, phn) attributes of an input tuple t match the (AC, Hphn) values of a master tuple s, and if $t[\mathsf{AC}, \mathsf{phn}]$ are validated, then $t[\mathsf{str}] := s[\mathsf{str}]$ (resp. city and zip). These eRs have a pattern $t[\mathsf{type}] = 1$, *i.e.,* phn is home phone.
- ◦ φ_9 states that when the AC value of an input tuple t is not 0800 (toll free, non-geographic), if it agrees with a master tuple s on its AC attribute, and moreover, if $t[\mathsf{AC}]$ has been validated, then t should be updated with $t[\mathsf{city}] := s[\mathsf{city}]$. As shown in Fig. 3.10, the pattern "$\neq$ 0800" can be edited via a pop-up frame.

CERFIX automatically tests whether the specified eRs make sense *w.r.t.* master data, *i.e.,* the rules do not contradict each other and will lead to a *unique* fix for any input tuple. Furthermore, given certain attributes that are validated, it automatically derives what other attributes can be validated by eR and master data, via an inference system.

Data monitor. We show how CERFIX interacts with users to find a certain fix for each input tuple.

1. CERFIX suggests a set of attributes for the users to validate. The users may either



Input Data

FN	LN	AC	phn	type	str	city	zip	item
Robert	Brady	131	6884563	1	51 Elm Row	Edi	EH7 4AH	CD
Robert	Brady	131	0791724851	2	51 Elm Row	Edi	EH7 4AH	CD
Mark	Smith	201	0755684832	2	20 Baker St.	Ldn	NW1 6XE	DVD
Catherine	Smith	201	6884563	1	20 Baker St.	Ldn	NW1 6XE	BOOK

Page 1 of 1 10 View 1 - 5 of 5

Master Data

FN	LN	AC	Hphn	Mphn	str	city	zip	DOB	gender
Mark	Smith	201	6884563	07556848	20 Baker St.	Ldn	NW1 6XE	25/12/67	M

Editing Rules

id	LHS	RHS	Pattern
4	([phn], [Mphn])	(FN, FN)	View

Created by root on Fri Mar 25 09:45:51 GMT 2011
Repaired by CerFix on Fri Mar 25 09:46:41 GMT 2011

Figure 3.12: Data auditing.

validate these attributes, or opt to assure that some other attributes are correct. The initial suggestions are computed by region finder.

- As shown in Fig. 3.11(a), the attributes suggested by CERFIX are highlighted in yellow, *i.e.,* area code AC, phone number phn, phone type, and product item. The values of the attributes assigned by the users are 201, 075568485, Mobile phone (type 2), and DVD, respectively.

2. If the users opt to validate these attributes, CERFIX iteratively applies editing rules and master data to the data, and expands the set of attributes validated.

   - As shown in Fig. 3.11(b), all attributes that have been validated are now highlighted in green. These include attributes first name FN, last name LN, and city, for which the correctness is validated by CERFIX. For instance, the value of FN is normalized from 'M.' to 'Mark' by eR φ_4 with the FN value of the second master tuple in Fig. 3.10.

   The users may decide to validate attributes other than those suggested. CERFIX reacts by fixing data with editing rules and master data in the same way, based on the attributes selected and validated by the users.

3. If some attributes of the input tuple are still not validated, CERFIX computes a new suggestion and goes back to step 1, to interact with the users by providing

the new suggestion. In each interaction, both the users and CERFIX expand the attributes that are validated.

- ○ As shown in Fig. 3.11(b), CERFIX suggests the users to validate zip code. After two rounds of interactions, all the attributes are validated. This is shown in Fig. 3.11(c) with all attributes in green.

When fixing the data, the most time-consuming procedure is to compute suggestions. To reduce the cost, CERFIX pre-computes a set of certain regions with *region finder* (see Fig. 3.9), which are provided to the users as initial suggestions, and are referenced when computing new suggestions.

We remark that data monitor of CERFIX is quite generic, *i.e.,* it does not depend on any particular system. Indeed, it supports several interfaces to access data, which could be readily integrated with other database applications.

Data auditing. CERFIX provides a data auditing facility such that after a stream of input tuples is fixed, the users may inspect the changes made to those tuples.

The users may inspect attributes of an individual tuple. For instance, as shown in Fig. 3.12, when the users select the FN attribute of a tuple (highlighted in yellow), CERFIX shows that it has been fixed by normalizing the first name 'M.' to 'Mark'. It further presents what master tuples and editing rules have been employed to make the change.

The users may also want to inspect each attribute (column) of the input tuples. As shown in Fig. 3.12, when FN is selected, CERFIX presents the statistics about the attribute FN, namely, the percentage of FN values that were validated by the users and the percentage of values that were automatically fixed by CERFIX. Our experimental study indicates that in average, 20% of values are validated by users while CERFIX automatically fixes 80% of the data.

Summary. We exhibit the strength of editing rules and important functionalities of CERFIX. (1) Editing rules. As opposed to integrity constraints that only detect the presence of errors in the data, editing rules identify what attributes are erroneous and tell us how to correct the errors with master data. (2) Region finder. It tells us to validate an input tuple, what minimal sets of attributes have to be assured correct. (3) Data monitor. It interacts with the users to find certain fixes, while minimizing human efforts by suggesting a minimal number of attributes for the users to validate. (4) Data auditing. It helps the users understand better the quality of input data sets.

3.7 Experimental Study

We next present an experimental study, using real-life data. Two sets of experiments were conducted, to verify (1) the effectiveness of our method in terms of the quality of suggestions generated, measured by the number of attributes that are correctly fixed in a round of user interactions; and (2) the efficiency and scalability of our algorithm for finding fixes and suggestions.

For the effectiveness study, we compared with the following: (a) GRegion that greedily finds a certain region. It chooses attributes according to one rule: at each stage, choose an attribute which may fix the largest number of uncovered attributes; and (b) IncRep, the algorithm in [CFG$^+$07] for data repairing; given a dirty database D and a set of constraints, it is a heuristic method to make D consistent, *i.e.,* finds a repair D' that satisfies the constraints and "minimally" differs from D. It adopts a metric to minimize (1) the distance between the original values and the new values of changed attributes and (2) the weights of the attributes modified.

Experimental data. Real-life datasets were employed to examine the applicability of our method in practice.

(1) HOSP *(Hospital Compare)* is publicly available from U.S. Department of Health & Human Services*. We used three tables: HOSP, HOSP_MSR_XWLK, and STATE_MSR_AVG, which record the hospital information, the score of measurement of each hospital and the average score of each hospital measurement, respectively. We created a big table by joining the three tables with *natural join*, among which we chose 19 attributes for the schema of both the master relation R_m and the relation R: zip, ST (state), phn, mCode (measure code), measure name, sAvg (StateAvg), hName (hospital name), hospital type, hospital owner, provider number, city, emergency service, condition, Score, sample, id, address1, address2, address3.

We designed 21 eRs for the HOSP data, with five representative ones as follows:

φ_1 : ((zip, zip) $\rightarrow$ (ST, ST), t_{p1}[zip] = $(\overline{\text{nil}})$);

φ_2 : ((phn, phn) $\rightarrow$ (zip, zip), t_{p2}[phn] = $(\overline{\text{nil}})$);

φ_3 : (((mCode, ST), (mCode, ST)) $\rightarrow$ (sAvg,sAvg), t_{p3} = ());

φ_4 : (((id, mCode), (id, mCode)) $\rightarrow$ (Score,Score), t_{p4} = ());

φ_5 : ((id, id) $\rightarrow$ (hName,hName), t_{p5} = ()).

(2) DBLP is from the DBLP Bibliography†. We first transformed the XML data into

*http://www.hospitalcompare.hhs.gov/

†http://www.informatik.uni-trier.de/∼ley/db/

relations. We then created a big table by joining the *inproceedings* data (conference papers) with the *proceedings* data (conferences) on the crossref attribute (a foreign key). Besides, we also included the homepage info (hp) for authors, which was joined by the homepage entries in the DBLP data.

From the big table, we chose 12 attributes to specify the schema of both the master relation R_m and the data relation R, including ptitle (paper title), a1 (the first author), a2 (the second author), hp1 (the homepage of a1), hp2 (the homepage of a2), btitle (book title), publisher, isbn, crossref, year, type, and pages.

We designed 16 eRs for the DBLP data, shown below.

$\phi_1 : ((\mathsf{a1}, \mathsf{a1}) \rightarrow (\mathsf{hp1}, \mathsf{hp1}), t_{p1}[\mathsf{a1}] = (\overline{\text{nil}}));$

$\phi_2 : ((\mathsf{a2}, \mathsf{a1}) \rightarrow (\mathsf{hp2}, \mathsf{hp1}), t_{p2}[\mathsf{a2}] = (\overline{\text{nil}}));$

$\phi_3 : ((\mathsf{a2}, \mathsf{a2}) \rightarrow (\mathsf{hp2}, \mathsf{hp2}), t_{p3}[\mathsf{a2}] = (\overline{\text{nil}}));$

$\phi_4 : ((\mathsf{a1}, \mathsf{a2}) \rightarrow (\mathsf{hp1},\mathsf{hp2}), t_{p4}[\mathsf{a1}] = (\overline{\text{nil}}));$

$\phi_5 : (((\mathsf{type}, \mathsf{btitle}, \mathsf{year}), (\mathsf{type}, \mathsf{btitle}, \mathsf{year})) \rightarrow$
$\quad (\text{A}, \text{A}), t_{p5}[\mathsf{type}] = (\text{`inproceeding'}));$

$\phi_6 : (((\mathsf{type}, \mathsf{crossref}), (\mathsf{type}, \mathsf{crossref}) \rightarrow$
$\quad (\text{B}, \text{B}), t_{p6}[\mathsf{type}] = (\text{`inproceeding'}));$

$\phi_7 : (((\mathsf{type}, \mathsf{a1}, \mathsf{a2}, \mathsf{title}, \mathsf{pages}), (\mathsf{type}, \mathsf{a1}, \mathsf{a2}, \mathsf{title}, \mathsf{pages})) \rightarrow$
$\quad (\text{C}, \text{C}), t_{p7}[\mathsf{type}] = (\text{`inproceeding'})).$

where the attributes A, B and C range over the sets $\{\mathsf{isbn}, \mathsf{publisher}, \mathsf{crossref}\}$, $\{\mathsf{btitle}, \mathsf{year}, \mathsf{isbn}, \mathsf{publisher}\}$ and $\{\mathsf{isbn}, \mathsf{publisher}, \mathsf{year}, \mathsf{btitle}, \mathsf{crossref}\}$, respectively.

Observe that in eRs ϕ_2 and ϕ_4, the attributes are mapped to different attributes. That is, even when the master relation R_m and the relation R share the same schema, some eRs still could not be syntactically expressed as CFDs, not to mention their semantics.

A dirty data generator was developed. Given a clean dataset (HOSP or DBLP), it generated dirty data controlled by three parameters: (a) duplicate rate $d\%$, which is the probability that an input tuple matches a tuple in master data D_m, indicating the relevance and completeness of D_m; (b) noise rate $n\%$, which is the percentage of erroneous attributes in input tuples; and (c) the cardinality $|D_m|$ of master dataset D_m.

User interactions. User feedback was simulated by providing the correct values of the given suggestions.

Implementation. All algorithms were implemented in C++. The experiments were run on a machine with an Intel(R) Core(TM)2 Duo P8700 (2.53GHz) CPU and 4GB of memory. Each experiment was repeated 5 times and the average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness. The tests were conducted by varying $d\%$, $|D_m|$ and $n\%$, The default values for $d\%$, $|D_m|$ and $n\%$ were 30%, 10K and 20%, respectively. When all these parameters were fixed, we generated 10K tuples for this set of experiments, but allowed the dataset to scale to 10M tuples in the scalability study.

This set of experiments includes (1) the effectiveness of certain regions generated by our algorithm compared with GRegion; (2) the initial suggestion selection; (3) the effectiveness of suggestions in terms of the number of interaction rounds needed; (4) the impact of duplicate rate $d\%$; (5) the impact of master data size $|D_m|$; (6) the impact of noise rate $n\%$; and (7) the effectiveness of our method compared with IncRep.

The studies were quantified at both the tuple level and the attribute level. Since we assure that each fixed tuple is correct, we have a 100% precision. Hence the first measure we used is *recall*, defined as follows:

$$\text{recall}_t = \text{\#-corrected tuples} \;/\; \text{\#-erroneous tuples}$$
$$\text{recall}_a = \text{\#-corrected attributes} \;/\; \text{\#-erroneous attributes}$$

The number of corrected attributes does *not* include those fixed by the users.

To compare with IncRep, we also used F-measure[‡] to combine *recall* and *precision*, since the precision of repairs produced by IncRep is not 100%. Precision and F-measure are given as follows:

$$\text{precision}_a = \text{\#-corrected attributes} \;/\; \text{\#-changed attributes}$$
$$\text{F-measure} = 2 \cdot (\text{recall}_a \cdot \text{precision}_a)/(\text{recall}_a + \text{precision}_a)$$

(1) The effectiveness of certain regions. The table below shows the number of attributes in the certain region found by our method CompCRegion [FLM+10] and its counterpart found by GRegion. It shows that the certain region computed by CompCRegion has far less attributes than its counterpart by GRegion, which thus minimizes user efforts, as expected. Indeed, CompCRegion found the best certain region (*i.e.,* with the least number of attributes) for both datasets as a suggestion.

Dataset	CompCRegion	GRegion
HOSP	2	4
DBLP	5	9

(2) The initial suggestion selection. We evaluated the impact of initial suggestions by using the certain region with the highest quality (denoted by CRHQ) vs. the one with

[‡] http://en.wikipedia.org/wiki/F-measure

the median quality (CRMQ). As shown in the table below, when CRHQ is used as the initial suggestion, CertainFix yields higher F-measure values than its CRMQ counterpart. That is, CRHQ allows CertainFix to automatically fix more attributes than CRMQ.

Dataset	**F-measure**	
	CRHQ	CRMQ
HOSP	0.74	0.70
DBLP	0.79	0.69

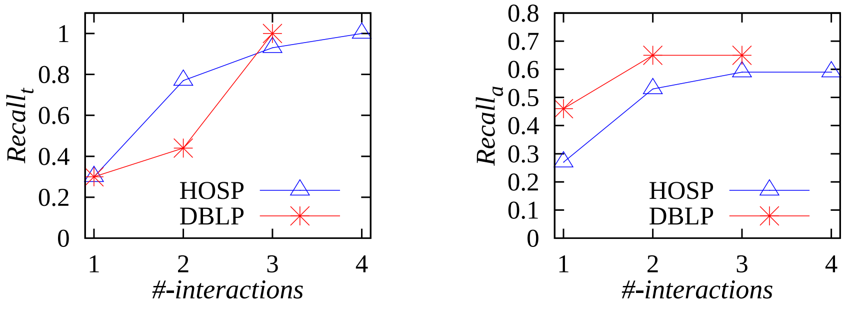
(3) The effectiveness of suggestions. Fixing the three parameters, we studied recall *w.r.t.* user interactions.

Figure 3.13(a) shows the tuple-level recalls. The x-axis indicates the number of interactions and the y-axis represents recall values. It tells us that few rounds of interactions are required to fix the entire set of attributes of an input tuple, *e.g.,* at most 4 (resp. 3) rounds for HOSP (resp. DBLP). Most tuples could be correctly fixed within few interactions, *e.g.,* 93% (resp. 100%) of tuples are fixed in the third round for HOSP (resp. DBLP).

Figure 3.13(b) reports the attribute-level recalls, to complement Figure 3.13(a). Among the errors fixed, some were automatically corrected by our algorithm, while the others by user feedback during the interactions. As remarked earlier, the errors fixed by the users were *not* counted in our recall values. Hence recall_a is typically below 100%. As shown in Fig. 3.13(b), our method could fix at least 50% of the errors within 2 rounds of interactions, although the errors were distributed across all attributes, and moreover, only a portion of the errors were fixable by the given Σ and D_m given that the duplicate rate $d\%$ is only 30%. One can see that the recall value at the 4th (resp. 3rd) round of interaction for HOSP (resp. DBLP) is unchanged, indicating that the users corrected the attributes that are irrelevant to Σ and D_m. As will be seen later, when $d\%$ is increased, the attribute-level recall gets higher.

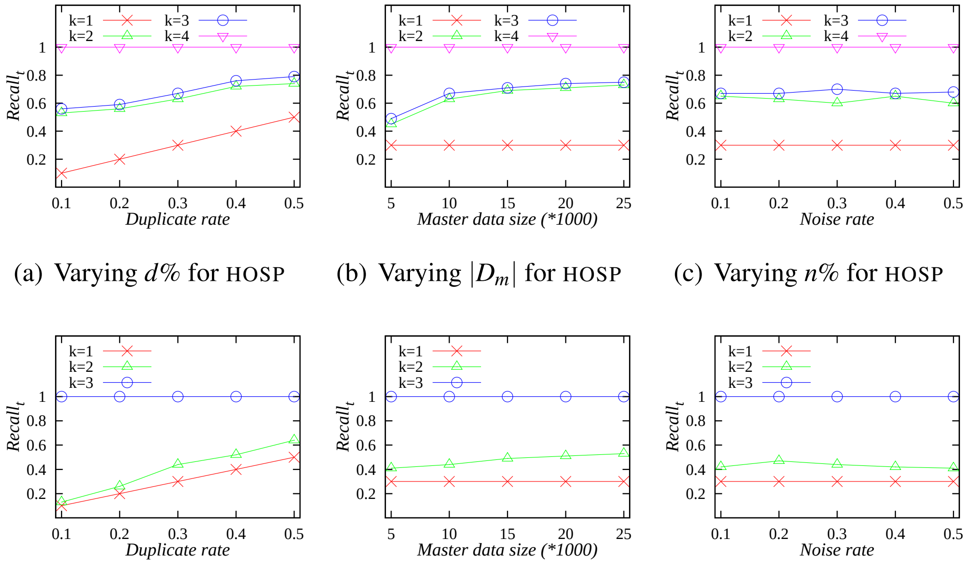
These experimental results verify that our method is able to provide effective suggestions, such that all errors could be fixed within few rounds of user interactions, by using eRs and master data, even when the master data is not very relevant (when $d\%$ = 30%).

(4) Impact of $d\%$. Fixing $|D_m| = 10K$ and $n\% = 20\%$, we varied duplicate rate $d\%$ from 10% to 50%. Figures 3.14(a) and 3.14(d) (resp. Figures 3.15(a) and 3.15(d)) report the



(a) Tuple-level recalls (b) Attribute-level recalls

Figure 3.13: Recall values *w.r.t.* the number of interactions



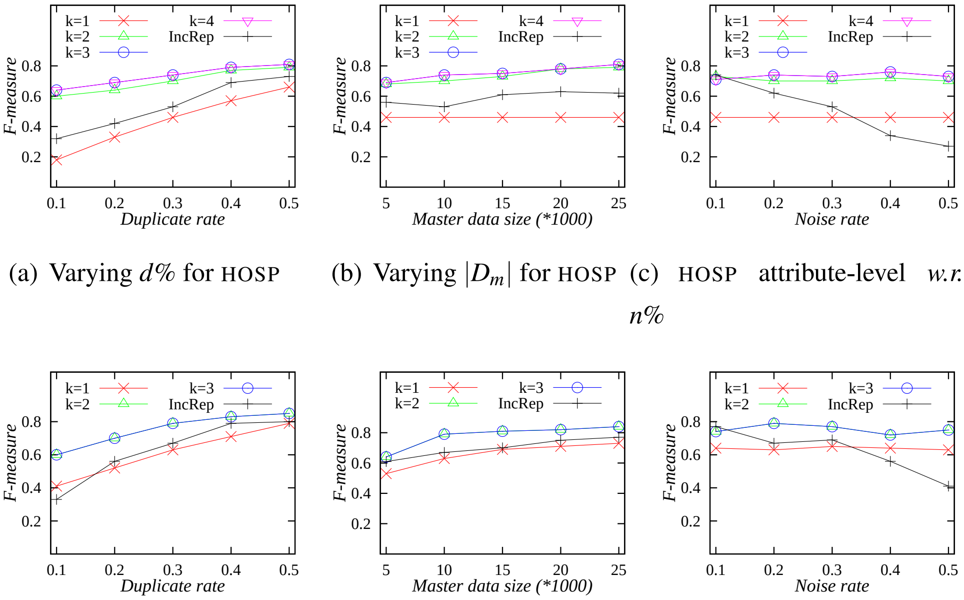
(a) Varying $d\%$ for HOSP (b) Varying $|D_m|$ for HOSP (c) Varying $n\%$ for HOSP

(d) Varying $d\%$ for DBLP (e) Varying $|D_m|$ for DBLP (f) Varying $n\%$ for DBLP

Figure 3.14: Tuple-level fixes when varying one of $d\%$, $|D_m|$ and $n\%$

tuple-level recalls (resp. F-measure) after k rounds of interactions for HOSP and DBLP, respectively.

Figures 3.14(a) and 3.14(d) show that the larger $d\%$ is, the higher the recall is, as expected, since a larger $d\%$ means a higher probability that an input tuple matches some master tuple such that its errors can be fixed. A closer examination reveals that early interactions are more sensitive to $d\%$, *e.g.,* when $k = 1$, the percentage of fixed tuples increases from 0.1 to 0.5, when $d\%$ varies from 10% to 50%. In later interactions, *e.g.,* the last round when $k = 4$, the users have to ensure the correctness of those attributes that cannot be fixed by eRs and D_m. Hence recall$_t$ remains unchanged there.



(a) Varying $d\%$ for HOSP (b) Varying $|D_m|$ for HOSP (c) HOSP attribute-level *w.r.t.* $n\%$

(d) Varying $d\%$ for DBLP (e) Varying $|D_m|$ for DBLP (f) DBLP attribute-level *w.r.t.* $n\%$

Figure 3.15: Attribute-level fixes when varying one of $d\%$, $|D_m|$ and $n\%$

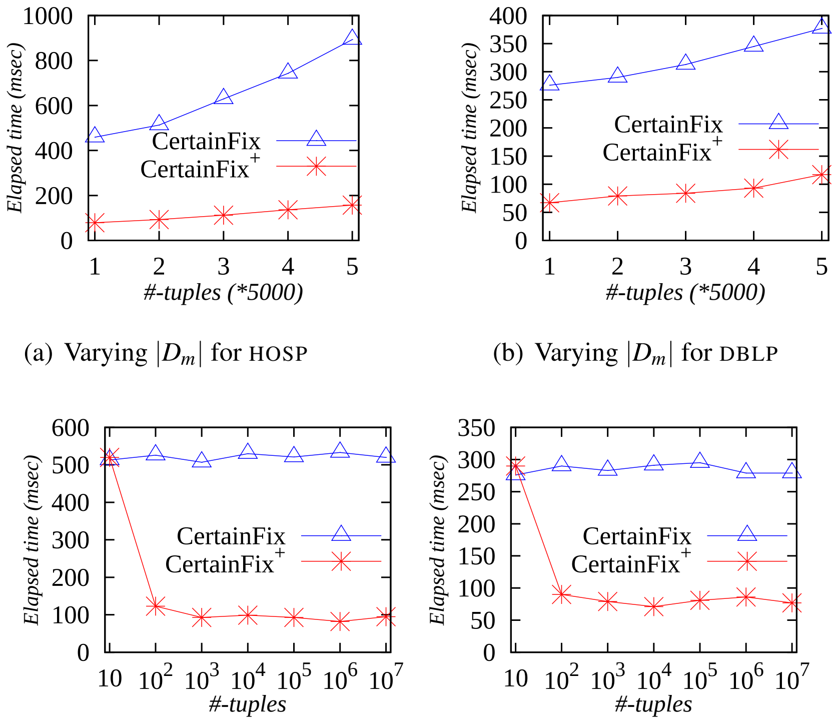
Figures 3.15(a) and 3.15(d) further verify this observation: most attributes are fixed by our method in early interactions, while those fixed in later rounds are by the users' feedback. Moreover, the gap between the first two rounds of interactions (when $k = 1$ and $k = 2$) shows that the suggestions generated are effective.

The results tell us that our method is sensitive to duplicate rate $d\%$: the higher $d\%$ is, the more errors could be automatically fixed, in early interactions.

(5) Impact of $|D_m|$. Fixing $d\% = 30\%$ and $n\% = 20\%$, we varied $|D_m|$ from $5K$ to $25K$. The tuple-level recalls (resp. F-measure values) are reported in Figures 3.14(b) and 3.14(e) (resp. Figures 3.15(b) and 3.15(e)) after k rounds of interactions for HOSP and DBLP, respectively.

Figures 3.14(b) and 3.14(e) show that in the first round of interactions, *i.e.,* $k = 1$, recall$_t$ is insensitive to $|D_m|$. Indeed, whether a certain fix exists or not in the first interaction is determined by the duplicate rate $d\%$, rather than $|D_m|$. As shown in both figures, the recall$_t$ is 0.3 when $k = 1$, exactly the same as $d\%$. However, when interacting with the users, the recall values increase for larger D_m. This verifies that TransFix is effective, which identifies eRs and master data to fix errors.

Figures 3.15(b) and 3.15(e) show that more attributes can be fixed by increasing $|D_m|$, *i.e.,* F-measure gets higher, even when the recall$_t$ is unchanged (*e.g.,* $k = 1$), *i.e.,*



(a) Varying $|D_m|$ for HOSP

(b) Varying $|D_m|$ for DBLP

(c) Varying $|D|$ for HOSP

(d) Varying $|D|$ for DBLP

Figure 3.16: Efficiency and Scalability

when not the entire tuple could be fixed. These results also confirm the observations above about the sensitivity of later rounds of interactions to $|D_m|$.

These results tell us that the amount of master data is important to generating effective suggestions. The more the master data, the higher possibility that eRs could find master tuples to fix attributes, as expected.

(6) Impact of $n\%$. Fixing $d\% = 30\%$ and $|D_m| = 10K$, we varied the noise rate $n\%$ from 0.1 to 0.5. Figures 3.14(c) and 3.14(f) (resp. Figures 3.15(c) and 3.15(f)) show the tuple-level recalls (resp. F-measure) after k rounds of interactions for HOSP and DBLP, respectively.

The results show that our method is sensitive to $n\%$ at *neither* the tuple level *nor* the attribute level. At the tuple level (Figures 3.14(c) and 3.14(f)), recall$_t$ is the ratio of the number of corrected tuples to the number of erroneous tuples. For a set of attributes asserted by the users, the attributes fixed by our algorithm remain the same for all input tuples, irrelevant to what attributes are originally erroneous. At the attribute-level (Figures 3.15(c) and 3.15(f)), since the precision of our algorithm is 100%, F-measure is determined by the recall values. As recall$_t$ is insensitive to $n\%$, so is F-measure.

(7) Comparison with IncRep. To favor IncRep, we fixed $k = 1$, since IncRep does not interact with the users. Since IncRep measures recall at the attribute level only [CFG$^+$07], we focus on F-measure. Figures 3.15(a) and 3.15(d) (resp. Figures 3.15(b) and 3.15(e)) show the F-measure values when varying $d\%$ (resp. $|D_m|$) while fixing the other two parameters. The results tell us that IncRep has slightly higher F-measure values than our method. This is because IncRep attempts to repair the entire tuple, while our method only corrects those attributes when the fixes are certain in the first round of interaction, and defers the repairing of the other attributes to later rounds upon the availability of user feedback.

Figures 3.15(c) and 3.15(f) show that when the noise rate $n\%$ is increased, the F-measure values of IncRep get substantially lower, and are worse than ours. This is because IncRep introduces more errors when the noise rate is higher. Our method, in contrast, ensures that each fix is correct, and hence is *insensitive* to $n\%$.

Exp-2: Efficiency and scalability. This set of experiments evaluated the efficiency of our method by varying the size of D_m (resp. a set D of input tuples) in Fig. 3.16(a) and Fig. 3.16(a) for HOSP (resp. Fig. 3.16(b) and Fig. 3.16(d) for DBLP). We report the average elapsed time for each round of interaction, *i.e.,* the time spent on fixing tuples in D and for generating a suggestion. Here CertainFix and CertainFix$^+$ denotes the algorithm that does not use BDD and employs BDD, respectively.

Figures 3.16(a) and 3.16(b) show that our method takes no more than a second to fix attributes of a tuple and to come up with a suggestion. Further, the optimization strategy by using BDD is effective: it substantially reduces the response time. Moreover, both CertainFix and CertainFix$^+$ scale well with master data.

As shown in Figures 3.16(c) and 3.16(d), CertainFix is insensitive to $|D|$, since each input tuple is processed independently. For CertainFix$^+$, when $|D|$ is very small (*e.g.,* 10), BDD does not help us find suggestions, and the elapsed time of CertainFix $^+$ is similar to the time of CertainFix; when $|D|$ increases from 10 to 100, the response time is significantly reduced since more suggestions could be found with BDD; when $|D| > 100$, BDD can provide effective suggestions such that the average elapsed time remains unchanged, around 0.1 second.

Summary. The experimental results show the followings. (1) The initial suggestions computed by our method are more effective than those found by greedy approaches. (2) Our method is effective: it mostly takes less than four rounds of user interactions to find a certain fix for an input tuple. (3) The number of interactions highly depends on the relevance of an input tuple to the master data, *i.e.,* $d\%$, and $|D_m|$ to a lesser extent. (4) Our method is insensitive to the error rate $n\%$. It outperforms the repairing method of [CFG$^+$07] when the error rate is high, even with two or three rounds of interactions. (5) Our algorithm scales well with the size of D_m. (6) The optimization strategy with BDD is effective in finding suggestions with low latency.

It should be remarked that data monitoring incurs extra overhead of fixing input tuples for the database engine. Nevertheless, as pointed out by [SMO07], it is far less costly to correct a tuple at the point of data entry than fixing it afterward. The need for this is particularly evident when it comes to critical data. In addition, as verified by our experimental results, the extra cost is rather small since effective suggestions (Exp-1 (1-3)) and certain fixes (Exp-2) can be generated efficiently, below 0.2 second in average with CertainFix$^+$ (Fig. 12).

Chapter 4

Interaction Between Record Matching and Data Repairing

4.1 Introduction

In this chapter, we present our study on interaction between record matching and data repairing, the two central tasks of data cleaning.

Many data cleaning tools in the market support record matching, and some also support the functionality of data repairing. But these systems treat matching and repairing as independent processes and perform them separately. However, the two processes typically interact with each other: matching helps us find repairs and repairing helps us identify matches, as shown below.

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

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

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

(b) Database D: An instance of schema tran

Figure 4.1: Example master data and database

Example 4.1.1: Consider two databases D_m and D from a UK bank: D_m maintains customer information collected when credit cards are issued, and is treated as *clean master data* [Los09]; D consists of transaction records of credit cards, which may be dirty. The databases are specified by schemas:

$$\mathsf{card}(\mathsf{FN}, \mathsf{LN}, \mathsf{St}, \mathsf{city}, \mathsf{AC}, \mathsf{zip}, \mathsf{tel}, \mathsf{dob}, \mathsf{gd}),$$
$$\mathsf{tran}(\mathsf{FN}, \mathsf{LN}, \mathsf{St}, \mathsf{city}, \mathsf{AC}, \mathsf{post}, \mathsf{phn}, \mathsf{gd}, \mathsf{item}, \mathsf{when}, \mathsf{where}).$$

Here a card tuple specifies a UK credit card holder identified by first name (FN), last name (LN), address (street (St), city, zip code), area code (AC), phone (tel), date of birth (dob) and gender (gd). A tran tuple is a record of a purchased item paid by a credit card at place where and time when, by a UK customer who is identified by name (FN, LN), address (St, city, post code), AC, phone (phn) and gender (gd). Example instances of card and tran are shown in Figures 4.1(a) and 4.1(b), which are fractions of D_m and D, respectively (the cf rows in Fig. 4.1(b) will be discussed later).

Following [FGJK08, FJLM09], we use conditional functional dependencies (CFDs [FGJK08]) φ_1–φ_4 to specify the consistency of tran data D, and a matching dependency (MD [FJLM09]) ψ as a rule for matching tuples across D and master card data D_m:

$$\varphi_1\text{: } \mathsf{tran}([\mathsf{AC} = 131] \rightarrow [\mathsf{city} = \text{Edi}]),$$
$$\varphi_2\text{: } \mathsf{tran}([\mathsf{AC} = 020] \rightarrow [\mathsf{city} = \text{Ldn}]),$$
$$\varphi_3\text{: } \mathsf{tran}([\mathsf{city}, \mathsf{phn}] \rightarrow [\mathsf{St}, \mathsf{AC}, \mathsf{post}]),$$
$$\varphi_4\text{: } \mathsf{tran}([\mathsf{FN} = \text{Bob}] \rightarrow [\mathsf{FN} = \text{Robert}]),$$
$$\psi\text{: } \mathsf{tran}[\mathsf{LN}, \mathsf{city}, \mathsf{St}, \mathsf{post}] = \mathsf{card}[\mathsf{LN}, \mathsf{city}, \mathsf{St}, \mathsf{zip}] \wedge$$
$$\mathsf{tran}[\mathsf{FN}] \approx \mathsf{card}[\mathsf{FN}] \rightarrow \mathsf{tran}[\mathsf{FN}, \mathsf{phn}] \rightleftharpoons \mathsf{card}[\mathsf{FN}, \mathsf{tel}],$$

where (1) CFD φ_1 (resp. φ_2) asserts that if the area code is 131 (resp. 020), the city must be Edi (resp. Ldn); (2) CFD φ_3 is a traditional functional dependency (FD) asserting that city and phone number uniquely determine street, area code and postal code; (3) CFD φ_4 is a data standardization rule: if the first name is Bob, then it should be "normalized" as Robert; and (4) MD ψ assures that for any tuple in D and any tuple in D_m, if they have the same last name and address, and moreover, if their first names are *similar*, then their phone and FN attributes can be identified.

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

Observe that t_3 and t_4 are quite different in their $\mathsf{FN}, \mathsf{city}$, $\mathsf{St}, \mathsf{post}$ and Phn attributes. No rule allows us to identify the two directly. Nonetheless, they can indeed be matched by a sequence of *interleaved* matching and repairing operations:

(a) get a repair t'_3 of t_3 such that $t'_3[\mathsf{city}] = \text{Ldn}$ via CFD φ_2, and $t'_3[\mathsf{FN}]$=Robert by normalization with φ_4;

(b) match t'_3 with s_2 of D_m, to which ψ can be applied;

(c) as a result of the matching operation, get a repair t''_3 of t_3 by correcting $t''_3[\mathsf{phn}]$ with the master data $s_2[\mathsf{tel}]$;

(d) find a repair t'_4 of t_4 via the FD φ_3: since t''_3 and t_4 agree on their city and phn attributes, φ_3 can be applied. This allows us to enrich $t_4[\mathsf{St}]$ and fix $t_4[\mathsf{post}]$ by taking corresponding values from t''_3, which have been confirmed correct with the master data in step (c).

At this point t''_3 and t'_4 agree on every attribute in connection with personal information. It is now evident enough that they indeed refer to the same person; hence a fraud.

Observe that not only repairing helps matching (*e.g.,* from step (a) to (b)), but matching also helps us repair the data (*e.g.,* step (d) is doable only after the matching in (b)). □

From this example, we observe the following. (1) When working together, record matching and data repairing perform much better than being treated as independent processes. (2) To make practical use of their interaction, matching and repairing operations should be *interleaved*. It does not help much to execute these processes consecutively one after another.

There has been many works on record matching (*e.g.,* [ARS09, BSIBD09, CGGM03, FJLM09, HS98, WBGM09]; see [EIV07, HSW09a] for surveys) as well as on data repairing (*e.g.,* [ABC03, BFFR05, CFG$^+$07, FLM$^+$10, FH76, MNP10, YENO10]). However, to the best of our knowledge, no previous work has studied the interaction between record matching and data repairing.

A unified process for repairing and matching is practical, and that it should logically become part of data cleaning systems.

While master data is desirable in the process, it is not a must. Indeed, in its absence, our approach can be adapted by interleaving (a) record matching in a single data table with MDs, as described in [FJLM09], and (b) data repairing with CFDs. While deterministic fixes may have lower accuracy, reliable and heuristic fixes would not degrade substantially.

Related work. Record matching is also known as record linkage, entity resolution, and duplicate detection [ARS09, BSIBD09, CGGM03, FJLM09, HS98, WBGM09, WN05, DHM05, GDSZ10] (see [HSW09a, EIV07] for surveys). Matching rules are studied in [FJLM09, HS98] (positive) and [ARS09, WBGM09] (negative). Data repairing was first studied in [ABC03, FH76]. A variety of constraints have been used to specify data consistency in data repairing, *e.g.,*FDs [Wij05], FDs and INDs [BFFR05], and CFDs [CFG$^+$07, FGJK08]. We employ CFDs, and extend MDs of [FJLM09] with negative rules.

The consistency and implication problems have been studied for CFDs [FGJK08] and MDs [FJLM09]. We study these problems for MDs and CFDs put together. It is known that data repairing is NP-complete [BFFR05, CFG$^+$07]. We show that data cleaning via repairing and matching is NP-complete and approximation-hard. We also study the termination and determinism analyses of data cleaning, which are not considered in [BFFR05, CFG$^+$07].

Several repairing algorithms have been proposed [BFFR05, CFG$^+$07, FLM$^+$10, FH76, MNP10, YENO10]. Heuristic methods are developed in [BFFR05, CFG$^+$07, FH76], based on FDs and INDs [BFFR05], CFDs [FGJK08], and edit rules [FH76]. The methods of [BFFR05, CFG$^+$07] employ confidence placed by users to guide a repairing process. Statistical inference is studied in [MNP10] to derive missing values. To ensure the accuracy of repairs generated, [MNP10, YENO10] require to consult users. In contrast to the previous work, we (a) unify repairing and matching, (b) use confidence just to derive deterministic fixes, and (c) leverage master data and entropy to improve the accuracy. Closer to our work is [FLM$^+$10], also based on master data. It differs from our work in the following. (i) While [FLM$^+$10] aims to fix a *single* tuple via matching with editing rules (derived from MDs), we repair a *database* via *both* matching (MDs) and repairing (CFDs), a task far more challenging. (ii) While [FLM$^+$10] only relies on confidence to warrant the accuracy, we use entropy analysis when the confidence is either low or unavailable.

There have also been efforts to interleave merging and matching operations [WN05, DHM05, WBGM09, GDSZ10]. Among these, (1) [GDSZ10] proposes to use uniqueness constraints to cluster objects from multiple data sources, and employs machine learning techniques to discover the true values of the objects; it differs from this work in the set of constraints used; and (2) [WN05, DHM05, WBGM09] investigate record matching in the presence of error data, and advocate the need for data repairing to match records. The merge/fusion operations adopted there are more restrictive than updates (value modifications) suggested by cleaning rules of this work. Furthermore, when no matches are found, no merge or fusion can be conducted, whereas this work may still repair data with CFDs.

There has also been a host of work on ETL tools (see [HSW09a] for a survey), which support data transformations, and can be employed to merge and fix data [NBBW06], although they are typically not based on a constraint theory. These are essentially complementary to data repairing and this work.

Information entropy measures the degree of uncertainty [CT91]: the less the entropy is, the more certain the data is. It has proved effective in, *e.g.,* database design, schema matching, data anonymization and data clustering [SV10]. We make a first effort to use it in data cleaning: we mark a fix reliable if its entropy is below a predefined threshold.

Organization. Section 4.2 reviews CFDs and extends MDs. Section 4.3 introduces the framework for data cleaning. Section 4.4 studies the fundamental problems for data cleaning. Algorithms for finding deterministic and reliable fixes are provided in Sections 4.5 and 4.6, respectively. Section 4.7 reports our experimental study.

4.2 Data Quality Rules

Below we first review CFDs [FGJK08], which specify the consistency of data for data repairing. We then extend MDs [FJLM09] to match tuples across (a possibly dirty) database D and master data D_m. Both CFDs and MDs can be automatically discovered from data via profiling algorithms (*e.g.,* [CM08, SC09]).

4.2.1 Conditional Functional Dependencies

Following [FGJK08], we define *conditional functional dependencies* (CFDs) on a relation schema R as follows.

As depicted in Chapter 2, a CFD φ defined on schema R is a pair $R(X \to Y,\, t_p)$, where (1) $X \to Y$ is a standard FD on R, referred to as *the FD embedded in* φ; and (2) t_p is a *pattern tuple* with attributes in X and Y, where for each A in $X \cup Y$, $t_p[A]$ is either a constant in the domain $\mathsf{dom}(A)$ of attribute A, or an unnamed variable '_' that draws values from $\mathsf{dom}(A)$.

We separate the X and Y attributes in t_p with '$\|$', and refer to X and Y as the LHS and RHS of φ, respectively.

Example 4.2.1: Recall the CFDs φ_1, φ_3 and φ_4 given in Example 4.1. These can be formally expressed as follows.

φ_1: $\mathsf{tran}([\mathsf{AC}] \to [\mathsf{city}], t_{p_1} = (131 \parallel \text{Edi}))$,
φ_3: $\mathsf{tran}([\mathsf{city}, \mathsf{phn}] \to [\mathsf{St}, \mathsf{AC}, \mathsf{post}], t_{p_3} = (\_, \_ \parallel \_, \_, \_))$
φ_4: $\mathsf{tran}([\mathsf{FN}] \to [\mathsf{FN}], t_{p_4} = (\text{Bob} \parallel \text{Robert}))$

Note that FDs are a special case of CFDs in which pattern tuples consist of only wildcards, *e.g.,* φ_3 given above. □

To give the formal semantics of CFDs, we use an operator $\asymp$ defined on constants and '_': $v_1 \asymp v_2$ if either $v_1 = v_2$, or one of v_1, v_2 is '_'. The operator $\asymp$ naturally extends to tuples, *e.g.,* (131, Edi) $\asymp$ (_, Edi) but (020, Ldn) $\not\asymp$ (_, Edi).

Consider an instance D of R. We say that D *satisfies* the CFD φ, denoted by $D \models \varphi$, iff for *all* tuples t_1, t_2 in D, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$.

Example 4.2.2: Recall the tran instance D of Fig. 4.1(b) and the CFDs of Example 4.2.1. Observe that $D \not\models \varphi_1$ since tuple $t_1[\mathsf{AC}] = t_{p_1}[\mathsf{AC}]$, but $t_1[\mathsf{city}] \neq t_{p_1}[\mathsf{city}]$, *i.e.,* the single tuple t_1 violates φ_1. Similarly, $D \not\models \varphi_4$, as t_3 does not satisfy φ_4. Intuitively, φ_4 says that no tuple t can have $t[\mathsf{FN}]$ = Bob (it has to be changed to Robert). In contrast, $D \models \varphi_3$: there exist no distinct tuples in D that agree on city and phn. □

We say that an instance D of R *satisfies* a set Σ of CFDs, denoted by $D \models \Sigma$, if $D \models \varphi$ for each $\varphi \in \Sigma$.

4.2.2 Positive and Negative Matching Dependencies

Following [FJLM09, HS98], we define matching dependencies (MDs) in terms of a set Υ of similarity predicates, *e.g.,* q-grams, Jaro distance or edit distance (see *e.g.,* [EIV07] for a survey).

We define positive MDs and negative MDs across a data relation schema R and a master relation schema R_m.

Positive MDs. A positive MD ψ on (R, R_m) is defined as:

$$\bigwedge_{j\in[1,k]}(R[A_j] \approx_j R_m[B_j]) \to \bigwedge_{i\in[1,h]}(R[E_i] \rightleftharpoons R_m[F_i]),$$

where (1) for each $j \in [1,k]$, A_j and B_j are attributes of R and R_m, respectively, with the same domain; similarly for E_i and F_i ($i \in [1,h]$); and (2) $\approx_j$ is a similarity predicate in Υ that is defined in the domain of $R[A_j]$ and $R_m[B_j]$. We refer to $\bigwedge_{j\in[1,k]}(R[A_j] \approx_j R_m[B_j])$ and $\bigwedge_{i\in[1,h]}(R[E_i] \rightleftharpoons R_m[F_i])$ as the LHS (premise) and RHS of ψ, respectively.

Note that MDs were originally defined on one or more unreliable data sources (see [FJLM09] for a detailed discussion of their dynamic semantics). In contrast, we focus on matching tuples across a dirty source D and a master relation D_m. To cope with this, we refine the semantics of MDs as follows.

For a tuple $t \in D$ and a tuple $s \in D_m$, if for each $j \in [1,k]$, $t[A_j]$ and $s[B_j]$ are similar, *i.e.,* $t[A_j] \approx_j s[B_j]$, then $t[E_i]$ is *changed* to $s[F_i]$, the clean master data, for each $i \in [1,h]$.

We say that an instance D of R *satisfies* the MD ψ *w.r.t. master data* D_m, denoted by $(D, D_m) \models \psi$, iff for *all* tuples t in D and *all* tuples s in D_m, if $t[A_j] \approx_j s[B_j]$ for $j \in [1,k]$, then $t[E_i] = s[F_i]$ for all $i \in [1,h]$.

Intuitively, $(D, D_m) \models \psi$ if no more tuples from D can be matched (and hence updated) with master tuples in D_m.

Example 4.2.3: Recall MD ψ given in Example 4.1.1. Consider an instance D_1 of tran consisting of a single tuple t'_1, where $t'_1[\mathsf{city}]$ = Ldn and $t'_1[A] = t_1[A]$ for all the other attributes, for t_1 given in Fig. 4.1(b). Then $(D_1, D_m) \not\models \psi$, since $t'_1[\mathsf{FN},\mathsf{phn}] \neq s_1[\mathsf{FN},\mathsf{tel}]$ while (t'_1[LN, city, St, post] = $s_1[\mathsf{LN},\mathsf{city},\mathsf{St},\mathsf{Zip}]$ and $t'_1[\mathsf{FN}] \approx s_1[\mathsf{FN}]$. This suggests that we correct $t'_1[\mathsf{FN},\mathsf{phn}]$ using the master data $s_1[\mathsf{FN},\mathsf{tel}]$. □

Negative MDs. Along the same lines as [ARS09, WBGM09], we define a negative MD ψ^- as follows:

$$\bigwedge_{j\in[1,k]}(R[A_j] \;\neq\; R_m[B_j]) \to \bigvee_{i\in[1,h]}(R[E_i] \not\rightleftharpoons R_m[F_i]).$$

It states that for any tuple $t \in D$ and any tuple $s \in D_m$, if $t[A_j] \neq s[B_j]$ ($j \in [1,k]$), then t and s may not be identified.

Example 4.2.4: A negative MD defined on $(\mathsf{tran},\mathsf{card})$ is:

$$\psi_1^-\!: \mathsf{tran}[\mathsf{gd}] \neq \mathsf{card}[\mathsf{gd}] \to \bigvee_{i\in[1,7]}(\mathsf{tran}[A_i] \not\rightleftharpoons \mathsf{card}[B_i]),$$

where (A_i, B_i) ranges over $(\mathsf{FN},\mathsf{FN})$, $(\mathsf{LN},\mathsf{LN})$, $(\mathsf{St},\mathsf{St})$, $(\mathsf{AC},\mathsf{AC})$, $(\mathsf{city},\mathsf{city})$, $(\mathsf{post},\mathsf{zip})$ and $(\mathsf{phn},\mathsf{tel})$. It says that a male and a female may not refer to the same person. □

We say that an instance D of R *satisfies* the negative MD ψ^- *w.r.t. master data* D_m, denoted by $(D, D_m) \models \psi^-$, if for *all* tuples t in D and *all* tuples s in D_m, if $t[A_j] \neq s[B_j]$ for all $j \in [1,k]$, then there exists $i \in [1,h]$ such that $t[E_i] \neq s[F_i]$.

An instance D of R *satisfies* a set Γ of (positive, negative) MDs *w.r.t. master data* D_m, denoted by $(D, D_m) \models \Gamma$, if $(D, D_m) \models \psi$ for all $\psi \in \Gamma$.

Normalized CFDs and MDs. Given a CFD (resp. MD) ξ, we use $\text{LHS}(\xi)$ and $\text{RHS}(\xi)$ to denote the LHS and RHS of ξ, respectively. It is called *normalized* if $|\text{RHS}(\xi)| = 1$, *i.e.,* its right-hand side consists of a single attribute (resp. attribute pair). As shown by [FGJK08, FJLM09], every CFD ξ (resp. MD) can be expressed as an equivalent set S_ξ of CFDs (resp. MDs), such that the cardinality of S_ξ is bounded by the size of $\text{RHS}(\xi)$.

For instance, CFDs φ_1, φ_2 and φ_4 of Example 4.1.1 are normalized. While φ_3 is not normalized, it can be converted to an equivalent set of CFDs of the form $([\mathsf{city}, \mathsf{phn}] \to A_i, t_{p_i})$, where A_i ranges over St, AC and post, and t_{p_i} consists of wildcards only; similarly for MD ψ.

We consider normalized CFDs (MDs) only in the sequel.

4.3 A Uniform Framework for Data Cleaning

We propose a rule-based framework for data cleaning. It treats CFDs and MDs uniformly as *cleaning rules*, which tell us how to fix errors, and seamlessly interleaves matching and repairing operations (Section 4.3.1). Using cleaning rules we introduce a tri-level data cleaning solution, which generates fixes with various levels of accuracy, depending on the information available about the data (Section 4.3.2).

Consider a (possibly dirty) relation D of schema R, a master relation D_m of schema R_m, and a set $\Theta = \Sigma \cup \Gamma$, where Σ is a set of CFDs on R, and Γ is a set of MDs on (R, R_m).

4.3.1 A Rule-based Logical Framework

We first state the data cleaning problem, and then define cleaning rules derived from CFDs and MDs.

Data cleaning. Following [ABC03], we state the *data cleaning problem*, referred to as DCP, as follows. It takes D, D_m and Θ as input, and computes a *repair* D_r of D, *i.e.,* another database such that (a) $D_r \models \Sigma$, (b) $(D_r, D_m) \models \Gamma$, and (c) $\mathsf{cost}(D_r, D)$ is minimum. Intuitively, (a) D_r should be *consistent*, (b) no more tuples in D_r can be *matched* to master data, and (c) D_r is accurate and is close to the original data D. Following [CFG+07], we define $\mathsf{cost}(D_r, D)$ as:

$$\sum_{t \in D} \sum_{A \in \mathsf{attr}(R)} t(A).\mathsf{cf} * \frac{\mathsf{dis}_A(t[A], t'[A])}{\mathsf{max}(|t[A]|, |t'[A]|)}$$

where (a) tuple $t' \in D_r$ is the repair of tuple $t \in D$, (b) $\mathsf{dis}_A(v, v')$ is the distance between values $v, v' \in \mathsf{dom}(A)$; the smaller the distance is, the closer the two values are to each other; (c) $|t[A]|$ denotes the size of $t[A]$; and (d) $t[A].\mathsf{cf}$ is the *confidence* placed by the user in the accuracy of the attribute $t[A]$ (see the cf rows in Fig. 4.1(b)).

This quality metric says that *the higher the confidence* of the attribute $t[A]$ is and *the more distant* v' is from v, the more costly the change is. Thus, *the smaller* $\mathsf{cost}(D_r, D)$ is, *the more accurate and closer to* the original data D_r is. We use $\mathsf{dis}(v, v')/\mathsf{max}(|v|, |v'|)$ to measure the similarity of v and v' to ensure that longer strings with 1-character difference are closer than shorter strings with 1-character difference.

As remarked in [CFG+07], confidence can be derived via provenance analysis, which can be reinforced by recent work on determining the reliability of data sources (*e.g.,* [DBEHS10]).

Cleaning rules. A variety of integrity constraints have been studied for data repairing (*e.g.,* [BFFR05, CFG+07, FGJK08, Wij05]). As observed by [FLM+10], while there constraints help us determine whether data is dirty or not, *i.e.,* whether errors are present in the data, they do not tell us how to correct the errors.

To make better practical use of constraints in data cleaning, we define *cleaning rules*, which tell us what attributes should be updated and to what value they should be changed. From each MD in Γ and each CFD in Σ, we derive a cleaning rule as follows, based on fuzzy logic [KF88].

(1) MDs. Consider an MD $\psi = \bigwedge_{j\in[1,k]}(R[A_j] \approx_j R_m[B_j]) \to (R[E] \rightleftharpoons R_m[F])$. The *cleaning rule* derived from ψ, denoted by γ_ψ, *applies* a master tuple $s \in D_m$ to a tuple $t \in D$ if $t[A_j] \approx_j s[B_j]$ for each $j \in [1,k]$. It *updates* t by letting (a) $t[E] := s[F]$ and (b) $t[C].\mathsf{cf} := d$ for each $C \in E$, where d is the minimum $t[A_j].\mathsf{cf}$ for all $j \in [1,k]$ if $\approx_j$ is '='.

That is, γ_ψ corrects $t[E]$ with clean master value $s[F]$, and infers the new confidence of $t[E]$ following fuzzy logic [KF88].

(2) Constant CFDs. Consider a CFD $\varphi_c = R(X \to A,\ t_{p_1})$, where $t_{p_1}[A]$ is a *constant*. The *cleaning rule* derived from φ_c *applies to* a tuple $t \in D$ if $t[X] \asymp t_{p_1}[X]$ but $t[A] \neq$

$t_{p_1}[A]$. It *updates* t by letting (a) $t[A] := t_{p_1}[A]$, and (b) $t[A]$.cf $= d$, where d is the minimum $t[A']$.cf for all $A' \in X$. That is, the rule corrects $t[A]$ with the constant in the CFD.

(3) Variable CFDs. Consider a CFD $\varphi_v = (Y \to B, t_{p_2})$, where $t_{p_2}[B]$ is a wildcard '_'. The *cleaning rule* derived from φ_v is used to *apply* a tuple $t_2 \in D$ to another tuple $t_1 \in D$, where $t_1[Y] = t_2[Y] \asymp t_{p_2}[Y]$ but $t_1[B] \neq t_2[B]$. It *updates* t_1 by letting (a) $t_1[B] := t_2[B]$, and (b) $t_1[B]$.cf be the minimum $t_1[B']$.cf and $t_2[B']$.cf for all $B' \in Y$.

While cleaning rules derived from MDs are similar to editing rules of [FLM$^+$10], rules derived from (constant or variables) CFDs are not studied in [FLM$^+$10]. We use confidence information and infer new confidences based on fuzzy logic [KF88].

Embedding negative MDs. Recall negative MDs from Section 4.2.2. The example below tells us that negative MDs can be converted to equivalent positive MDs. As a result, there is no need to treat them separately.

Example 4.3.1: Consider MD ψ in Example 4.1.1 and negative MD ψ^- in Example 4.2.4. We define ψ' by incorporating the premise (gd) of ψ^- into the premise of ψ:

$$\begin{aligned}\psi'\colon\ & \mathsf{tran}[\mathsf{LN},\mathsf{city},\mathsf{St},\mathsf{post},\mathsf{gd}] = \mathsf{card}[\mathsf{LN},\mathsf{city},\mathsf{St},\mathsf{zip},\mathsf{gd}] \wedge \\ & \mathsf{tran}[\mathsf{FN}] \approx \mathsf{card}[\mathsf{FN}] \to \mathsf{tran}[\mathsf{FN},\mathsf{phn}] \rightleftharpoons \mathsf{card}[\mathsf{FN},\mathsf{tel}].\end{aligned}$$

Then no tuples with different genders can be identified as the same person, which is precisely what ψ^- is to enforce. In other words, the positive MD ψ' is equivalent to the positive MD ψ and the negative MD ψ^-. □

Indeed, it suffices to consider only positive MDs.

Proposition 4.3.1: *Given a set Γ_m^+ of positive MDs and a set Γ_m^- of negative MDs, there exists an algorithm that computes a set Γ_m of positive MDs in $O(|\Gamma_m^+||\Gamma_m^-|)$ time such that Γ_m is equivalent to $\Gamma_m^+ \cup \Gamma_m^-$.* □

A uniform framework. By treating both CFDs and MDs as cleaning rules, one can uniformly interleave matching and repairing operations, to facilitate their interactions.

Example 4.3.2: As shown in Example 4.1.1, to clean tuples t_3 and t_4 of Fig. 4.1(b), one needs to interleave matching and repairing operations. These can be readily done by using cleaning rules derived from φ_2, φ_4, ψ and φ_3. Indeed, the cleaning process described in Example 4.1.1 is actually carried out by applying these rules. There is no need to distinguish between matching and repairing in the cleaning process. □

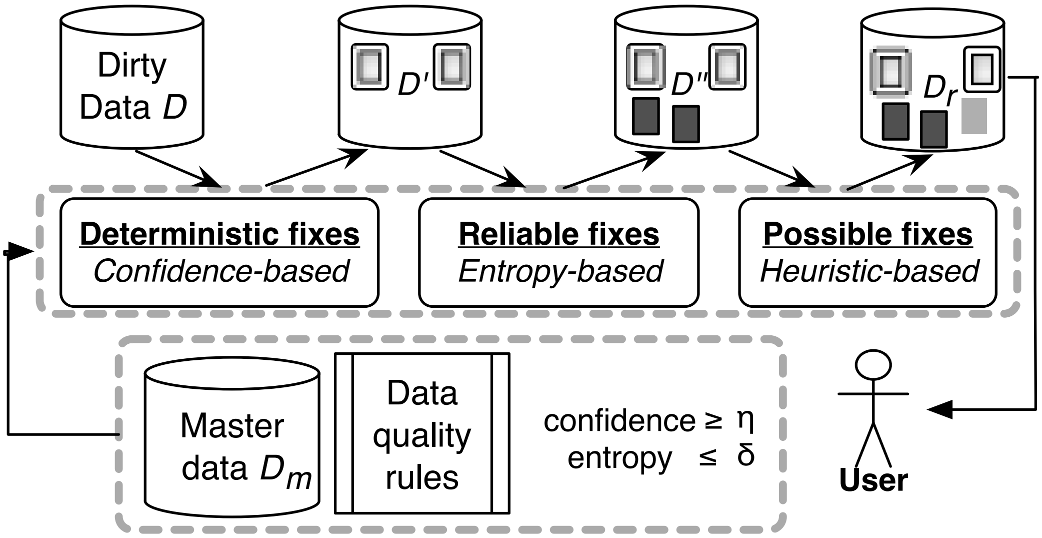


Figure 4.2: Framework Overview

4.3.2 A Tri-level Data Cleaning Solution

Based on cleaning rules, we develop a data cleaning system UniClean. It takes as input a dirty relation D, a master relation D_m, a set of cleaning rules derived from Θ, as well as thresholds $\eta, \delta \in [0,1]$ set by the users for confidence and entropy, respectively. It generates a repair D_r of D with a small $\mathsf{cost}(D_r, D)$, such that $D_r \models \Sigma$ and $(D_r, D_m) \models \Gamma$.

As opposed to previous repairing systems [BFFR05, CFG$^+$07, FLM$^+$10, FH76, MNP10, YENO10], UniClean generates fixes by unifying matching and repairing, via cleaning rules. Further, it stresses the accuracy by distinguishing these fixes with three levels of accuracy. Indeed, various fixes are found by three algorithms executed one after another, as shown in Fig. 4.2 and illustrated below.

(1) Deterministic fixes based on confidences. The first algorithm identifies erroneous attributes $t[A]$ to which there exists a unique fix, referred to as a *deterministic fix*, when some attributes of t are accurate. It fixes those errors based on confidence: it uses a cleaning rule to update $t[A]$ *only if* certain attributes of t have confidence above the threshold η. It is evident that such fixes are accurate up to η.

(2) Reliable fixes based on entropy. For attributes with *low or unavailable confidence*, we correct them based on the relative certainty of the data, measured by entropy. Entropy has proved effective in data transmission [Ham50] and compression [ZL78],

among other things. We use entropy to clean data: we apply a cleaning rule γ to update an erroneous attribute $t[A]$ only if the entropy of γ for certain attributes of t is below the threshold δ. Fixes generated via entropy are accurate to a certain degree, and are marked as *reliable fixes*.

(3) Possible fixes. Not all errors can be fixed in the first two phases. For the remaining errors, we adopt heuristic methods to generate fixes, referred to as *possible fixes*. To this end we extend the method of [CFG^{+}07], by supporting cleaning rules derived from both CFDs and MDs. It can be verified that the heuristic method always finds a repair D_r of D such that $D_r \models \Sigma$, $(D_r, D_m) \models \Gamma$, while keeping all the deterministic fixes produced earlier *unchanged*

At the end of the process, fixes are marked with three distinct signs, indicating deterministic, reliable and possible, respectively. We shall present methods based on confidence and entropy in Sections 4.5 and 4.6, respectively. Due to the space constraints, we omit the algorithm for possible fixes, but encourage the reader to consult [CFG^{+}07] for details.

4.4 Fundamental Problems for Data Cleaning

We now investigate fundamental problems associated with data cleaning. We first study the consistency and implication problems for CFDs and MDs taken together, from which cleaning rules are derived. We then establish the complexity bounds of the data cleaning problem as well as its termination and determinism analyses. These problems are not only of theoretical interest, but are also important to the development of data cleaning algorithms. The main conclusion of this section is that data cleaning via matching and repairing is inherently difficult: all these problems are intractable.

Consider a relation D, a master data D_m, and a set $\Theta = \Sigma \cup \Gamma$ of CFDs and MDs, as stated in Section 4.3.

4.4.1 Reasoning about Data Quality Rules

There are two classical problems for data quality rules.

The *consistency problem* is to determine, given D_m and $\Theta = \Sigma \cup \Gamma$, whether there exists a nonempty instance D of R such that $D \models \Sigma$ and $(D, D_m) \models \Gamma$.

Intuitively, this is to determine whether the rules in Θ are dirty themselves. The practical need for the consistency analysis is evident: it does not make sense to derive cleaning rules from Θ before Θ is assured consistent itself.

We say that Θ *implies* another CFD (resp. MD) ξ, denoted by $\Sigma \models \xi$, if for any instance D of R, whenever $D \models \Sigma$ and $(D, D_m) \models \Gamma$, then $D \models \xi$ (resp. $(D, D_m) \models \xi$).

The *implication problem* is to determine, given D_m, Σ and another CFD (or MD) ξ, whether $\Sigma \models \xi$.

Intuitively, the implication analysis helps us find and remove redundant rules from Σ, *i.e.,* those that are a logical consequence of other rules in Σ, to improve performance.

These problems have been studied for CFDs and MDs separately. It is known that the consistency problem for MDs is trivial: any set of MDs is consistent [FJLM09]. In contrast, there exist CFDs that are inconsistent, and the consistency analysis of CFDs is NP-complete [FGJK08]. It is also known that the implication problem for MDs and CFDs is in quadratic time [FJLM09] and coNP-complete [FGJK08], respectively.

We show that these problems for CFDs and MDs put together have the same complexity as their CFDs counterparts. That is, adding MDs to CFDs does not make our lives harder.

Theorem 4.4.1: *For CFDs and MDs put together, the consistency problem is NP-complete, and the implication problem is coNP-complete (when ξ is either a CFD or an MD).* □

Proof. The upper bounds are verified by establishing a small model property. The lower bounds follow from the intractability for their CFD counterparts, a special case. □

In the rest of the chapter we consider only collections Σ of CFDs and MDs that are consistent.

4.4.2 Analyzing the Data Cleaning Problem

Recall the data cleaning problem (DCP) from Section 4.3.

Complexity bounds. One wants to know how costly it is to compute a repair D_r. Below we show that it is intractable to decide whether there exists D_r with $\mathsf{cost}(D_r, D)$ below a predefine bound. Worse still, it is infeasible in practice to find PTIME approximation algorithm with performance guarantee. Indeed, the problem is not even in APX, the class of problems that allow PTIME approximation algorithms with approximation ratio bounded by a constant.

Theorem 4.4.2: *(a) The data cleaning problem (DCP) is NP-complete. (b) Unless* P =

NP, for any constant ε*, there exists no* PTIME ε*-approximation algorithm for DCP.* □

Proof. (a) The upper bound is verified by giving an NP algorithm. The lower bound is by reduction from 3SAT [WP05]. (b) This is verified by reduction from 3SAT, using gap techniques [WP05]. Given any constant ε, we show that there exists an algorithm with approximation ratio ε for DCP iff there is a PTIME algorithm for deciding 3SAT. □

It is known that data repairing alone is NP-complete [CFG$^+$07]. Theorem 4.4.2 tells us that when matching with MDs is incorporated, the problem is intractable and approximation-hard.

Termination and determinism analyses. There are two natural questions about rule-based data cleaning methods such as the one proposed in Section 4.3. (a) The *termination problem* is to determine whether a rule-based process stops. That is, it reaches a *fixpoint*, such that no cleaning rules can be further applied. (b) The *determinism problem* asks whether all terminating cleaning processes end up with the same repair, *i.e.,* all of them reach a *unique* fixpoint.

The need for studying these problems is evident. A rule-based process is often *non-deterministic*: multiple rules can be applied at the same time. We want to know whether the output of the process is independent of the order of the rules applied. Worse, it is known that even for repairing only, a rule-based method may lead to an *infinite* process [CFG$^+$07].

Example 4.4.1: Consider the CFD φ_1 = $\mathsf{tran}([\mathsf{AC}] \to [\mathsf{city}], t_{p_1} = (131 \parallel \text{Edi}))$ given in Example 4.2.1, and another CFD φ_5 = $\mathsf{tran}([\mathsf{post}] \to [\mathsf{city}], t_{p_5} = (\text{EH8 9AB} \parallel \text{Ldn}))$. Consider D_1 consisting of a single tuple t_2 given in Fig. 4.1. Then a repairing process for D_1 with φ_1 and φ_5 may fail to terminate: it changes $t_2[\mathsf{city}]$ to Edi and Ldn back and forth. □

No matter how important, it is beyond reach in practice to find efficient solutions to these two problems.

Theorem 4.4.3: *The termination and determinism problems are both PSPACE-complete for rule-based data cleaning.* □

Proof. We verify the lower bound of these problems by reduction from the halting problem for linear bound automata, which is PSPACE-complete [AKVW93]. We show the upper bound by providing an algorithm for each of the two problems, which uses polynomial space in the size of input. □

Symbols	Semantics
$\Theta = \Sigma \cup \Gamma$	A set Σ of CFDs and a set Γ of MDs
η, δ_1, δ_2	Confidence threshold, update threshold, and entropy threshold, respectively
ρ	Selection operator in relational algebra
π	Projection operator in relational algebra
$\Delta(\bar{y})$	The set $\{t \mid t \in D, t[Y] = \bar{y}\}$ for each $\bar{y}$ in $\pi_Y(\rho_{Y \asymp t_p[Y]} D)$ *w.r.t.* CFD $(Y \to B,\, t_p)$

Table 4.1: Summary of notations

4.5 Deterministic Fixes with Data Confidence

As shown in Fig. 4.2, system UniClean first identifies deterministic fixes based on confidence analysis and master data. In this section we define deterministic fixes (Section 4.5.1), and present an efficient algorithm to find them (Section 4.5.2).

In Table 4.1 we summarize some notations to be used in this Section and Section 4.6, for the ease of reference.

4.5.1 Deterministic Fixes

We define deterministic fixes *w.r.t.* a *confidence threshold* η determined by domain experts. When η is high enough, *e.g.,* if it is close to 1, an attribute $t[A]$ is assured correct if $t[A].\mathrm{cf} \geq \eta$. We refer to such attributes as *asserted* attributes. Recall from Section 4.3 the definition of cleaning rules derived from MDs and CFDs. In the first phase of UniClean, we apply a cleaning rule γ to tuples in a database D only when the attributes in the premise (*i.e.,* LHS) of γ are all asserted. We say that a fix is *deterministic w.r.t.* γ and η if it is generated as follows, based on how γ is derived.

(1) From an MD $\psi = \bigwedge_{j\in[1,k]}(R[A_j] \approx_j R_m[B_j]) \to (R[E] \rightleftharpoons R_m[F])$. Suppose that γ applies a tuple $s \in D_m$ to a tuple $t \in D$, and generates a fix $t[E] := s[F]$ (see Section 4.3.1). Then the fix is *deterministic* if $t[A_j].\mathrm{cf} \geq \eta$ for all $j \in [1,k]$ and moreover, $t[E].\mathrm{cf} < \eta$. That is, $t[E]$ is changed to the master value $s[F]$ only if (a) all the premise attributes $t[A_j]$'s are asserted, and (b) $t[E]$ is not yet asserted.

(2) From a constant CFD $\varphi_c = R(X \to A,\, t_{p_1})$. Suppose that γ applies to a tuple $t \in D$ and changes $t[A]$ to the constant $t_{p_1}[A]$ in φ_c. Then the fix is *deterministic* if $t[A_i].\mathrm{cf} \geq \eta$

for all $A_i \in X$ and $t[A]$.cf $< \eta$.

(3) From a variable CFD $\varphi_v = (Y \to B,\ t_p)$. For each $\bar{y}$ in $\pi_Y(\rho_{Y \asymp t_p[Y]} D)$, we define $\Delta(\bar{y})$ to be the set $\{t \mid t \in D, t[Y] = \bar{y}\}$, where π and ρ are the projection and selection operators, respectively, in relational algebra [AHV95]. That is, for all t_1, t_2 in $\Delta(\bar{y})$, $t_1[Y] = t_2[Y] = \bar{y} \asymp t_p[Y]$.

Suppose that γ applies a tuple t_2 in $\Delta(\bar{y})$ to another t_1 in $\Delta(\bar{y})$ for some $\bar{y}$, and changes $t_1[B]$ to $t_2[B]$. Then the fix is *deterministic* if (a) for all $B_i \in Y$, $t_1[B_i]$.cf $\geq \eta$ and $t_2[B_i]$.cf $\geq \eta$, (b) $t_2[B]$.cf $\geq \eta$, and moreover, (c) t_2 is the only tuple in $\Delta(\bar{y})$ with $t_2[B]$.cf $\geq \eta$ (hence $t_1[B]$.cf $< \eta$). That is, all the premise attributes of γ are asserted, and $t_2[B]$ is the only value of B-attribute in $\Delta(\bar{y})$ that is assumed correct, while $t_1[B]$ is suspected erroneous.

As observed by [FLM$^+$10], when data quality rules and asserted attributes are assured correct, the fixes generated are unique (called "certain" in [FLM$^+$10]). While [FLM$^+$10] only considers MDs, the observation remains intact for CFDs and MDs.

Note that when an attribute $t[A]$ is updated by a deterministic fix, its confidence $t[A]$.cf is upgraded to be the minimum of the confidences of the premise attributes (see Section 4.3.1). As a result, $t[A]$ also becomes asserted, since all premise attributes have confidence values above η. In turn $t[A]$ can be used to generate deterministic fixes for other attributes in the cleaning process. In other words, the process for finding deterministic fixes in a database D is *recursive*.

Nevertheless, in the rest of the section we show that deterministic fixes can be found in PTIME, stated as follows.

Theorem 4.5.1: *Given master data D_m and a set Θ of CFDs and MDs, all deterministic fixes in a relation D can be found in $O(|D||D_m|\mathsf{size}(\Theta))$ time, where $\mathsf{size}(\Theta)$ is Θ's length.* □

4.5.2 Confidence-based Data Cleaning

We next present the algorithm, followed by the indexing structures and procedures that it employs.

Algorithm. The algorithm, denoted by cRepair, is shown in Fig. 4.3. It takes as input CFDs Σ, MDs Γ, master data D_m, dirty data D, and a confidence threshold η. It returns a partially cleaned repair D' with deterministic fixes marked.

Algorithm cRepair first initializes variables and indexing structures (lines 1–6).

It then recursively computes deterministic fixes (lines 7–15), by invoking procedures vCFDInfer (line 12), cCFDInfer (line 13), or MDInfer (line 14), for rules derived from variable CFDs, constant CFDs, or MDs, respectively. It checks each tuple at most once *w.r.t.* each rule, makes more attributes asserted at each step, and uses these attributes to identify more deterministic fixes. It terminates when no more deterministic fixes can be found (line 15). Finally, a partially cleaned database D' is returned in which all deterministic fixes are marked (line 16).

Indexing structures. The algorithm uses the following indexing structures, to improve performance.

Hash tables. We maintain a hash table for each variable CFD $\varphi = R(Y \to B, t_p)$, denoted as H_φ. Given a $\bar{y} \in \rho_{Y \asymp t_p[Y]}(D)$ as the key, it returns a pair $(\mathsf{list}, \mathsf{val})$ as the value, *i.e.,* $\mathsf{H}(\bar{y}) = (\mathsf{list}, \mathsf{val})$, where (a) list consists of all the tuples t in $\Delta(\bar{y})$ such that $t[B_i].\mathsf{cf} \geq \eta$ for each attribute $B_i \in Y$, and (b) val is $t[B]$ if it is the only item in $\Delta(\bar{y})$ with $t[B].\mathsf{cf} \geq \eta$; otherwise, val is nil. Notably, there exist no two t_1, t_2 in $\Delta(\bar{y})$ such that $t_1[B] \neq t_2[B]$, $t_1[B].\mathsf{cf} \geq \eta$ and $t_2[B].\mathsf{cf} \geq \eta$, if the confidence placed by the users is correct.

Queues. We maintain for each tuple t a queue of rules that can be applied to t, denoted as $\mathsf{Q}[t]$. More specifically, $\mathsf{Q}[t]$ contains all rules $\xi \in \Theta$, where $t[C].\mathsf{cf} \geq \eta$ for all attributes C in $\mathrm{LHS}(\xi)$. That is, the premise of ξ is asserted in t.

Hash sets. For each tuple $t \in D$, $\mathsf{P}[t]$ stores the set of variable CFDs $\varphi \in \mathsf{Q}[t]$ such that $\mathsf{H}_\varphi(t[\mathrm{LHS}(\varphi)]).\mathsf{val} = \mathsf{nil}$, *i.e.,* no B attribute in $\Delta(t[\mathrm{LHS}(\varphi)])$ has a high enough confidence.

Counters. For each tuple $t \in D$ and each rule $\xi \in \Theta$, $\mathsf{count}[t, \xi]$ maintains the number of current values of the attributes $C \in \mathrm{LHS}(\xi)$ such that $t[C].\mathsf{cf} \geq \eta$.

Procedures. We now present the procedures of cRepair.

update. Given a new deterministic fix for $t[A]$, it propagates the change, to find other deterministic fixes with $t[A]$. (a) For each rule ξ, if $A \in \mathrm{LHS}(\xi)$, $\mathsf{count}[t, \xi]$ is increased by 1 as one more attribute becomes asserted. (b) If all attributes in $\mathrm{LHS}(\xi)$ are asserted, ξ is inserted into the queue $\mathsf{Q}[t]$. (c) For a variable CFD $\xi' \in \mathsf{P}[t]$, if $\mathrm{RHS}(\xi')$ is A and $\mathsf{H}_{\xi'}(t[\mathrm{LHS}(\xi')]).\mathsf{val} = \mathsf{nil}$, the newly asserted $t[A]$ makes it possible for tuples in $\mathsf{H}_{\xi'}(t[\mathrm{LHS}(\xi')]).\mathsf{list}$ to have a deterministic fix. Thus ξ' is removed from $\mathsf{P}[t]$ and added to $\mathsf{Q}[t]$.

vCFDInfer. Given a tuple t, a variable CFD ξ and the confidence threshold η, it finds a deterministic fix for t by applying ξ if it exists. If the tuple t and the pattern tuple $t_{(p,\xi)}$

```
Algorithm cRepair

Input: CFDs $\Sigma$, MDs $\Gamma$, master data $D_m$, dirty data $D$, and
       confidence threshold $\eta$.
Output: A partial repair $D'$ of $D$ with deterministic fixes.

$D' := D$;     $\mathsf{H}_\xi := \emptyset$ for each variable CFD $\xi \in \Sigma$;
for each $t \in D'$ do
    $\mathsf{Q}[t] := \emptyset$;  $\mathsf{P}[t] := \emptyset$;
    $\mathsf{count}[t,\xi]$ :=0 for each $\xi \in \Sigma \cup \Gamma$;
    for each attribute $A \in \mathsf{attr}(\Sigma \cup \Gamma)$ do
        if $t[A].\mathsf{cf} \geq \eta$ then $\mathsf{update}(t,A)$;
repeat
   for each tuple $t \in D'$ do
      while $\mathsf{Q}[t]$ is not empty do
         $\xi := \mathsf{Q}[t].\mathsf{pop}()$;
         case $\xi$ of
         (1) variable CFD: $D' := \mathsf{vCFDInfer}(t, \xi, \eta)$;
         (2) constant CFD: $D' := \mathsf{cCFDInfer}(t, \xi, \eta)$;
         (3) MD:           $D' := \mathsf{MDInfer}(t, \eta, D_m, \xi)$;
until $\mathsf{Q}[t']$ is empty for any $t' \in D'$;
return $D'$.
```

Figure 4.3: Algorithm cRepair

match on their LHS(ξ) attributes, it does the following.

(a) If $t[\text{RHS}(\xi)].\mathsf{cf} \geq \eta$ and if no B-attribute value in $\mathsf{H}_\xi(t[\text{LHS}(\xi)]).\mathsf{list}$ is asserted, it takes $t[\text{RHS}(\xi)]$ as the B value in the set, and propagates the change via update.

(b) If $t[\text{RHS}(\xi)] < \eta$ but there is an asserted B-attribute value val in $\mathsf{H}_\xi(t[\text{LHS}(\xi)]).\mathsf{list}$, it makes a deterministic fix by $t[\text{RHS}(\xi)] := \mathsf{val}$, and propagates the change via update.

(c) If $t[\text{RHS}(\xi)] < \eta$ and there is no asserted B-attribute in $\mathsf{H}_\xi(t[\text{LHS}(\xi)]).\mathsf{list}$, no deterministic fix can be made yet, and t is added to $\mathsf{H}_\xi(t[\text{LHS}(\xi)]).\mathsf{list}$ and $\mathsf{P}[t]$, for later checking.

cCFDInfer *and* MDInfer. The first one takes as input a tuple t, a constant CFD ξ and the threshold η. The second one takes as input t, η, master data D_m and an MD ξ. They find deterministic fixes by applying the rules derived from ξ, as described earlier. The changes made are propagated by invoking procedure $\mathsf{update}(t, \mathsf{RHS}(\xi))$.

Example 4.5.1: Consider master data D_m and relation D of Fig. 4.1. Assume Θ consists of rules ξ_1, ξ_2 and ξ_3 derived from CFDs φ_1, φ_3 and MD ψ of Example 4.1.1, respectively. Let the threshold η be 0.8. Using Θ and D_m, cRepair finds deterministic fixes for $t_1, t_2 \in D$ *w.r.t.* η as follows.

(1) After initialization (lines 1–6), we have: (a) $\mathsf{H}_{\xi_2} = \emptyset$; (b) $\mathsf{Q}[t_1] = \{\xi_1\}$, $\mathsf{Q}[t_2] = \{\xi_2\}$; (c) $\mathsf{P}[t_1] = \mathsf{P}[t_2] = \emptyset$; and (d) $\mathsf{count}[t_1, \xi_1] = 1$, $\mathsf{count}[t_1, \xi_2] = 0$, $\mathsf{count}[t_1, \xi_3] = 3$, $\mathsf{count}[t_2, \xi_1] = 0$, $\mathsf{count}[t_2, \xi_2] = 2$, and $\mathsf{count}[t_2, \xi_3] = 2$.

(2) After $\xi_2 \in \mathsf{Q}[t_2]$ is checked (line 12), we have $\mathsf{Q}[t_2] = \emptyset$, $\mathsf{P}[t_2] = \{\xi_2\}$, and $\mathsf{H}_{\xi_2}(t_2[\mathsf{city}, \mathsf{phn}]) = (\{t_2\}, \mathsf{nil})$.

(3) After $\xi_1 \in \mathsf{Q}[t_1]$ is applied (line 13), $\mathsf{Q}[t_1] = \{\xi_3\}$, $\mathsf{count}[t_1, \xi_2] = 1$ and $\mathsf{count}[t_1, \xi_3] = 4$. This step finds a deterministic fix $t_1[\mathsf{city}]$:= Edi. It upgrades $t_1[\mathsf{city}]$.cf:=0.8.

(4) When $\xi_3 \in \mathsf{Q}[t_1]$ is used (line 14), it makes a deterministic fix $t_1[\mathsf{phn}] := s_1[\mathsf{tel}]$, and lets $t_1[\mathsf{phn}]$.cf = 0.8. Now we have $\mathsf{Q}[t_1] = \{\xi_2\}$ and $\mathsf{count}[t_1, \xi_2] = 2$.

(5) When $\xi_2 \in \mathsf{Q}[t_1]$ is used (line 14), it finds a deterministic fix by letting $t_2[\mathsf{St}] = t_1[\mathsf{St}]$:= 10 Oak St, and $t_2[\mathsf{St}]$.cf := 0.8. Now we obtain $\mathsf{Q}[t_1] = \emptyset$ and $\mathsf{P}[t_2] = \emptyset$.

(6) Finally, the process terminates since $\mathsf{Q}[t_1] = \mathsf{Q}[t_2] = \emptyset$.

Similarly, for tuples $t_3, t_4 \in D$, cRepair finds a deterministic fix by letting $t_3[\mathsf{city}]$:= Ldn and $t_3[\mathsf{city}]$.cf := 0.8. □

Suffix trees for similarity checking of MDs. For cleaning rules derived from MDs, we need to conduct *similarity checking*, to which traditional indexing techniques are not directly applicable. To cope with this, we develop a technique based on suffix trees [dVKCC09]. The measure of similarity adopted is the length of the *longest common substring* of two strings. Generalized suffix trees are built for the blocking process with all the strings in the active domain. When querying the k-most similar strings of v of length $|v|$, we can extract the subtree T of suffix tree that only contains branches related to v, containing at most $|v|^2$ nodes. We traverse T to find the k-most similar strings. In this way, we can identify k similar values from D_m in $O(k|v|^2)$ time, which reduces the search space from $|D_m|$ to a constant number k of tuples. Our experimental study verifies that the technique significantly improves the performance.

Complexity. Each tuple t in D is examined at most twice for each CFD in Σ, and is checked at most $|D_m|$ times for each MD, each tuple. Hence cRepair is in $O(|D||D_m|\mathsf{size}(\Sigma \cup \Gamma))$ time. With the optimization methods above, the time complexity of cRepair is reduced to $O(|D|\mathsf{size}(\Sigma \cup \Gamma))$.

4.6 Reliable Fixes with Information Entropy

Deterministic fixes may not exist for some attributes, *e.g.,* when their confidences are low or unreliable. To find accurate fixes for these attributes, UniClean looks for evidence from data itself *instead of confidence*, using entropy to measure the degree of certainty. Below we first define entropy for data cleaning (Section 4.6.1), and present an algorithm to find reliable fixes using entropy (Section 4.6.2). We then present an indexing structure underlining the algorithm (Section 4.6.3).

4.6.1 Measuring Certainty with Entropy

We start with an overview of the standard information entropy, and then define entropy for resolving conflicts.

Entropy. The entropy of a discrete random variable $\mathcal{X}$ with possible values $\{x_1, \ldots, x_n\}$ is defined as [CT91, SV10]:

$$\mathcal{H}(\mathcal{X}) = \Sigma_{i=1}^{n}(p_i * \log 1/p_i),$$

where p_i is the probability of x_i for $i \in [1,n]$. The entropy measures the degree of the certainty of the value of $\mathcal{X}$: when $\mathcal{H}(\mathcal{X})$ is sufficiently small, it is highly accurate that the value of $\mathcal{X}$ is the x_j having the largest probability p_j. The less $\mathcal{H}(\mathcal{X})$ is, the more accurate the prediction is.

Entropy for variable CFDs. We use entropy to resolve data conflicts. Consider a CFD $\varphi = R(Y \to B, t_p)$ defined on a relation D, where $t_p[B]$ is a wildcard. Note that a deterministic fix may not exist when, *e.g.,* there are t_1, t_2 in $\Delta(\bar{y})$ (see Table 4.1) such that $t_1[B] \neq t_2[B]$ but both have high confidence. Indeed, using the cleaning rule derived from φ, one may either let $t_1[B] := t_2[B]$ by applying t_2 to t_1, or let $t_2[B] := t_1[B]$ by applying t_1 to t_2.

To find an accurate fix, we define the entropy of φ for $Y = \bar{y}$, denoted by $\mathcal{H}(\varphi | Y = \bar{y})$, as

```
Algorithm eRepair

Input: CFDs $\Sigma$, MDs $\Gamma$, master data $D_m$, dirty data $D$,
       update threshold $\delta_1$, entropy threshold $\delta_2$.
Output: A partial repair $D'$ of $D$ with reliable fixes.
$O$ := the order of $\Sigma \cup \Gamma$, sorted via their dependency graph;
$D'$ := $D$;
repeat
    for ($i$ = 1; $i \leq |\Sigma \cup \Gamma|$; $i$++) do
        $\xi$ := the $i$-th rule in $O$;
        case $\xi$ of
        (1) variable CFD: $D'$ := vCFDReslove($D'$, $\xi$, $\delta_1$, $\delta_2$);
        (2) constant CFD: $D'$ := cCFDReslove($D'$, $\xi$, $\delta_1$);
        (3) MD:           $D'$ := MDReslove($D'$, $D_m$, $\xi$, $\delta_1$);
until there are no changes in $D'$;
return $D'$.
```

Figure 4.4: Algorithm eRepair

$$\mathcal{H}(\varphi|Y=\bar{y}) = \Sigma_{i=1}^{k}(\frac{\mathsf{cnt}_{YB}(\bar{y},b_i)}{|\Delta(\bar{y})|} * \log_k \frac{|\Delta(\bar{y})|}{\mathsf{cnt}_{YB}(\bar{y},b_i)}),$$

where (a) $k = |\pi_B(\Delta(\bar{y}))|$, the number of distinct B values in $\Delta(\bar{y})$, (b) for each $i \in [1,k]$, $b_i \in \pi_B(\Delta(\bar{y}))$, (c) $\mathsf{cnt}_{YB}(\bar{y},b_i)$ denotes the number of tuples $t \in \Delta(\bar{y})$ with $t[B] = b_i$, and (d) $|\Delta(\bar{y})|$ is the number of tuples in $\Delta(\bar{y})$.

Intuitively, we treat $\mathcal{X}(\varphi|Y=\bar{y})$ as a random variable for the value of the B attribute in $\Delta(\bar{y})$, with a set $\pi_B(\Delta(\bar{y}))$ of possible values. The probability for b_i to be the value is $p_i = \frac{\mathsf{cnt}_{YB}(\bar{y},b_i)}{|\Delta(\bar{y})|}$. When $\mathcal{H}(\varphi|Y=\bar{y})$ is small enough, it is highly accurate to resolve the conflict by letting $t[B] = b_j$ for all $t \in \Delta(\bar{y})$, where b_j is the one with the highest probability, *i.e.*, $\mathsf{cnt}_{YB}(\bar{y},b_j)$ is maximum among all $b_i \in \pi_B(\Delta(\bar{y}))$.

In particular, $\mathcal{H}(\varphi|Y=\bar{y}) = 1$ when $\mathsf{cnt}_{YB}(\bar{y},b_i) = \mathsf{cnt}_{BA}(\bar{y},b_j)$ for all distinct $b_i, b_j \in \pi_B(\Delta(\bar{y}))$. If $\mathcal{H}(\varphi|Y=\bar{y}) = 0$ for all $\bar{y} \in \pi_Y(\rho_{Y \preceq t_p[Y]}D)$, then $D \models \varphi$.

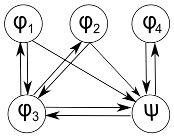


Figure 4.5: Example dependency graph

4.6.2 Entropy-based Data Cleaning

We first describe an algorithm based on entropy, followed by its main procedures and auxiliary structures.

Algorithm. The algorithm, referred to as eRepair, is shown in Fig. 4.4. Given a set Σ of CFDs, a set Γ of MDs, a master relation D_m, dirty data D, and two thresholds δ_1 and δ_2 for *update frequency* and *entropy*, respectively, it finds reliable fixes for D and returns a (partially cleaned) database D' in which reliable fixes are marked. The deterministic fixes found earlier by cRepair remain unchanged in the process.

In a nutshell, algorithm eRepair first sorts cleaning rules derived from the CFDs and MDs, such that rules with relatively bigger impact are applied early. Following the order, it then applies the rules one by one, until no more reliable fixes can be found. More specifically, it first finds an order O on the rules in $\Sigma \cup \Gamma$ (line 1). It then repeatedly applies the rules in the order O to resolve conflicts in D (lines 3–10), by invoking procedures vCFDReslove (line 7), cCFDReslove (line 8) or MDReslove (line 9), based on the types of the rules (lines 5-6). It terminates when either no more rules can be applied or all data values have been changed more than δ_1 times, *i.e.,* when there is no enough information to make reliable fixes (line 10). A partially cleaned database is returned with reliable fixes being marked (line 11).

Procedures. We next present the procedures of eRepair.

Sorting cleaning rules. To avoid unnecessary computation, we sort $\Sigma \cup \Gamma$ based on its *dependency graph* $G = (V, E)$. Each rule of $\Sigma \cup \Gamma$ is a node in V, and there is an edge from a rule ξ_1 to another ξ_2 if ξ_2 can be applied after the application of ξ_1. There exists an edge $(u, v) \in E$ from node u to node v if $\text{RHS}(\xi_u) \cap \text{LHS}(\xi_v) \neq \emptyset$. Intuitively, edge (u, v) indicates that whether ξ_v can be applied depends on the outcome of applying ξ_u. Hence, ξ_u should be applied before ξ_v. For instance, the dependency graph of the CFDs and MDs given in Example 4.1.1 is shown in Fig. 4.5.

Based on G, we sort the rules as follows. (1) Find strongly connected components (SCCs) in G, in linear time [CLRS01]. (2) By treating each SCC as a single node, we convert G into a DAG. (3) Find a topological order on the nodes in the DAG. That is, a rule ξ_1 is applied before another ξ_2 if the application of ξ_1 affects the application of ξ_2. (4) Finally, the nodes in each SCC are further sorted based on the ratio of its out-degree to in-degree, in a decreasing order. The higher the ratio is, the more effects it has on other nodes.

Example 4.6.1: The dependency graph G in Fig. 4.5 is an SCC. The ratios of out-degree to in-degree of the nodes φ_1, φ_2, φ_3, φ_4 and ψ are $\frac{2}{1}$, $\frac{2}{1}$, $\frac{1}{1}$, $\frac{3}{3}$ and $\frac{2}{4}$, respectively. Hence the order O of these rules is $\varphi_1 > \varphi_2 > \varphi_3 > \varphi_4 > \psi$, where those nodes with the same ratio are sorted randomly. □

vCFDReslove. It applies the cleaning rule derived from a variable CFD $\xi = R(Y \to B, t_p)$. For each set $\Delta(\bar{y})$ with $\bar{y}$ in $\pi_Y(\rho_{Y \asymp t_p[Y]} D)$, if $\mathcal{H}(\xi | Y = \bar{y})$ is smaller than the entropy threshold δ_2, it picks the value $b \in \pi_B(\Delta(\bar{y}))$ that has the maximum $\mathsf{cnt}_{YB}(\bar{y}, b)$. Then for each tuple $t \in \Delta(\bar{y})$, if $t[B]$ has been changed less than δ_1 times, *i.e.,* when $t[B]$ is not often changed by rules that may not converge on its value, $t[B]$ is changed to b. As remarked earlier, when the entropy $\mathcal{H}(\xi | Y = \bar{y})$ is small enough, it is highly accurate to resolve the conflicts in $\pi_B(\Delta(\bar{y}))$ by assigning b as their value.

cCFDReslove. It applies the rule derived from a constant CFD $\xi = R(X \to A, t_{p_1})$. For each tuple $t \in D$, if (a) $t[X] \asymp t_{p_1}[X]$, (b) $t[A] \neq t_{p_1}[A]$, and (c) $t[A]$ has been changed less than δ_1 times, then $t[A]$ is changed to the constant $t_{p_1}[A]$.

MDReslove. It applies the cleaning rule derived from an MD $\xi = \bigwedge_{j \in [1,k]} (R[A_j] \approx_j R_m[B_j]) \to R[E] \rightleftharpoons R_m[F]$. For each tuple $t \in D$, if there exists a master tuple $s \in D_m$ such that (a) $t[A_j] \approx_j s[B_j]$ for $j \in [1,k]$, (b) $t[E] \neq s[F]$, and (c) $t[E]$ has been changed less than δ_1 times, then it assigns the master value $s[F]$ to $t[E]$.

These procedures do not change those data values that are marked deterministic fixes by algorithm cRepair.

Example 4.6.2: Consider an instance of schema $R(\mathsf{ABCEFH})$ shown in Fig. 4.6, and a variable CFD $\phi = R(\mathsf{ABC} \to \mathsf{E}, t_{p_1})$, where t_{p_1} consists of wildcards only, *i.e.,* ϕ is an FD. Observe that (a) $\mathcal{H}(\phi | ABC = (a_1, b_1, c_1)) \approx 0.8$, (b) $\mathcal{H}(\phi | ABC = (a_2, b_2, c_2))$ is 1, and (c) $\mathcal{H}(\phi | ABC = (a_2, b_2, c_3))$ and $\mathcal{H}(\phi | ABC = (a_2, b_2, c_4))$ are both 0.

From these we can see the following. (1) For $\Delta(ABC = (a_2, b_2, c_3))$ and $\Delta(ABC = (a_2, b_2, c_4))$, the entropy is 0; hence these sets of tuples do not violate ϕ, *i.e.,* there is no need to fix these tuples. (2) The fix based on $\mathcal{H}(\phi | ABC = (a_1, b_1, c_1))$ is relatively accurate, but not those based on $\mathcal{H}(\phi | ABC = (a_2, b_2, c_2))$. Hence algorithm eRepair

	A	B	C	E	F	H
t_1:	a_1	b_1	c_1	e_1	f_1	h_1
t_2:	a_1	b_1	c_1	e_1	f_2	h_2
t_3:	a_1	b_1	c_1	e_1	f_3	h_3
t_4:	a_1	b_1	c_1	e_2	f_1	h_3
t_5:	a_2	b_2	c_2	e_1	f_2	h_4
t_6:	a_2	b_2	c_2	e_2	f_1	h_4
t_7:	a_2	b_2	c_3	e_3	f_3	h_5
t_8:	a_2	b_2	c_4	e_3	f_3	h_6

Figure 4.6: Example relation of schema R

will only change $t_4[E]$ to e_1, and marks it as a reliable fix. □

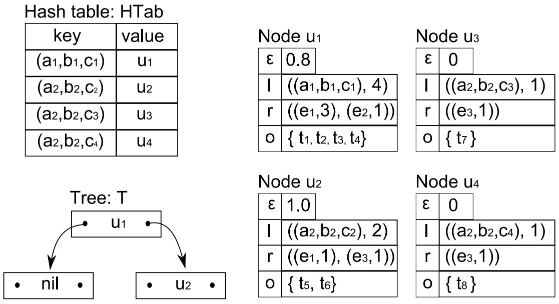
Complexity. The outer loop (lines 3–10) in algorithm eRepair runs in $O(\delta_1|D|)$ time. Each inner loop (lines 4–9) takes $O(|D||\Sigma| + k|D|\mathsf{size}(\Gamma))$ time using the optimization techniques of Section 4.5, where k is a constant. Thus, the algorithm takes $O(\delta_1|D|^2|\Sigma| + \delta_1 k|D|^2\mathsf{size}(\Gamma))$ time.

4.6.3 Resolving Conflicts with a 2-in-1 Structure

We can efficiently identify tuples that match the LHS of constant CFDs by building an index on the LHS attributes in the database D. We can also efficiently find tuples that match the LHS of MDs by leveraging the suffix tree structure developed in Section 4.5. However, for variable CFDs, two issues still remain: (a) detecting violations and (b) computing entropy. These are rather costly and have to be recomputed when data is updated in the cleaning process. To do these we develop a 2-in-1 structure, which can be easily maintained.

Let Σ_V be the set of variables CFDs in Σ, and $\mathsf{attr}(\Sigma_V)$ be the set of attributes appearing in Σ_V. For each CFD $\varphi = R(Y \to B, t_p)$ in Σ_V, we build a structure consisting of a *hash table* and an *AVL tree* [CLRS01] T as follows.

Hash table HTab. Recall $\Delta(\bar{y}) = \{t \mid t \in D,\ t[Y] = \bar{y}\}$ for $\bar{y} \in \pi_Y(\rho_{Y \asymp t_p[Y]} D)$ described earlier. For each $\Delta(\bar{y})$, we insert an entry $(\mathsf{key}, \mathsf{val})$ into HTab, where $\mathsf{key} = \bar{y}$, and val is a pointer linking to a node $\mathsf{u} = (\varepsilon, l, r, o)$, where (a) $\mathsf{u}.\varepsilon = \mathcal{H}(\varphi|Y = \bar{y})$, (b) $\mathsf{u}.l$ is the value-count pair $(\bar{y}, |\Delta(\bar{y})|)$, (c) $\mathsf{u}.r$ is the set $\{(b, \mathsf{cnt}_{YB}(\bar{y}, b)) \mid b \in \pi_B(\Delta(\bar{y}))\}$, and (d) $\mathsf{u}.o$ is the set of (partial) tuple IDs $\{t.id \mid t \in \Delta(\bar{y})\}$.



Hash table: HTab

key	value
(a_1,b_1,c_1)	u_1
(a_2,b_2,c_2)	u_2
(a_2,b_2,c_3)	u_3
(a_2,b_2,c_4)	u_4

Node u1

ε	0.8
l	$((a_1,b_1,c_1), 4)$
r	$((e_1,3), (e_2,1))$
o	$\{ t_1, t_2, t_3, t_4\}$

Node u3

ε	0
l	$((a_2,b_2,c_3), 1)$
r	$((e_3,1))$
o	$\{ t_7 \}$

Node u2

ε	1.0
l	$((a_2,b_2,c_2), 2)$
r	$((e_1,1), (e_3,1))$
o	$\{ t_5, t_6\}$

Node u4

ε	0
l	$((a_2,b_2,c_4), 1)$
r	$((e_3,1))$
o	$\{ t_8 \}$

Figure 4.7: Example data structure for variable CFDs

AVL tree T. For each $\bar{y} \in \pi_Y(\rho_{Y \asymp t_p[Y]} D)$ with entropy $\mathcal{H}(\varphi|Y = \bar{y}) \neq 0$, we create a node v = HTab($\bar{y}$) in T, a pointer to the node u for $\Delta(\bar{y})$ in HTab. For each node v in T, its left child $\mathsf{v}_l.\varepsilon \leq \mathsf{v}.\varepsilon$ and its right child $\mathsf{v}_r.\varepsilon \geq \mathsf{v}.\varepsilon$.

Note that both the number |HTab| of entries in the hash table HTab and the number $|T|$ of nodes in the AVL tree T are bounded by the number $|D|$ of tuples in D.

Example 4.6.3: Consider the relation in Fig. 4.6 and the variable CFD ϕ given in Example 4.6.2. The hash table HTab and the AVL tree T for ϕ are shown in Fig. 4.7. □

We next show how to use and maintain the structures.

(1) Lookup cost. For the CFD φ, it takes (a) $O(\log|T|)$ time to identify the set $\Delta(\bar{y})$ of tuples with minimum entropy $\mathcal{H}(\varphi|Y = \bar{y})$ in the AVL tree T, and (b) $O(1)$ time to check whether two tuples in D satisfy φ via the hash table HTab.

(2) Update cost. The initialization of both the hash table HTab and the AVL tree T can be done by scanning the database D once, and it takes $O(|D|\log|D||\Sigma_V|)$ time.

After resolving some conflicts, the structures need to be maintained accordingly. Consider a set $\Delta(\bar{y})$ of dirty tuples. When a reliable fix is found for $\Delta(\bar{y})$ based on $\mathcal{H}(\varphi|Y = \bar{y})$, we do the following: (a) remove a node from tree T, which takes $O(\log|T|)$ time, where $|T| \leq |D|$; and (b) update the hash tables and trees for all other CFDs, which takes $O(|\Delta(\bar{y})||\Sigma_V| + |\Delta(\bar{y})|\log|D|)$ time in total.

(3) Space cost. The structures take $O(|D|\mathsf{size}(\Sigma_V))$ space for all CFDs in Σ_V in total, where $\mathsf{size}(\Sigma_V)$ is the size of Σ_V.

Putting these together, the structures are efficient in both time and space, and are easy to maintain.

4.7 Experimental Study

We next present an experimental study of UniClean, which unifies matching and repairing. Using real-life data, we evaluated (1) the effectiveness of our data cleaning algorithms, (2) the accuracy of deterministic fixes and reliable fixes, and (3) the scalability of our algorithms with the size of data.

Experimental Setting. We used two real-life data sets.

(1) HOSP *data* was taken from US Department of Health & Human Services[*]. It has 100K records with 19 attributes. We designed 23 CFDs and 3 MDs for HOSP, 26 in total.

(2) DBLP *data* was extracted from DBLP Bibliography[†]. It consists of 400K tuples, each with 12 attributes. We designed 7 CFDs and 3 MDs for DBLP, 10 in total.

(3) Master data for both datasets was carefully selected from the same data sources so that they were guaranteed to be correct and consistent *w.r.t.* the designed rules.

(4) Dirty datasets were produced by introducing noises to data from the two sources, controlled by four parameters: (a) $|D|$: the data size; (b) noi%: the *noise rate*, which is the ratio of the number of erroneous attributes to the total number of attributes in D; (c) dup%: the *duplicate rate*, *i.e.,*, the percentage of tuples in D that can find a match in the master data; and (d) asr%: the *asserted rate*. For each attribute A, we randomly picked asr% of tuples t from the data and set $t[A].\mathsf{cf} = 1$, while letting $t'[A].\mathsf{cf} = 0$ for the other tuples t'. The default value for asr% is 40%.

Algorithms. We implemented the following algorithms, all in Python: (a) algorithms cRepair, eRepair and hRepair (an extension of algorithm in [CFG$^+$07]) in UniClean; (b) the sorted neighborhood method of [HS98], denoted by SortN, for record matching based on MDs only; and (c) the heuristic repairing algorithm of [CFG$^+$07], denoted by quaid, based on CFDs only. We use Uni to denote cleaning based on both CFDs and MDs (matching and repairing), and Uni(CFD) to denote cleaning using CFDs (repairing) only.

We used edit distance for similarity test, defined as the minimum number of single-character insertions, deletions and substitutions needed to convert a value from v to v'.

Quality measuring. We adopted *precision, recall* and *F-measure*, which are commonly used in information retrieval, where F-measure $= 2 \cdot (\text{precision} \cdot$

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

[†] http://www.informatik.uni-trier.de/∼ley/db/

recall)/(precision + recall).

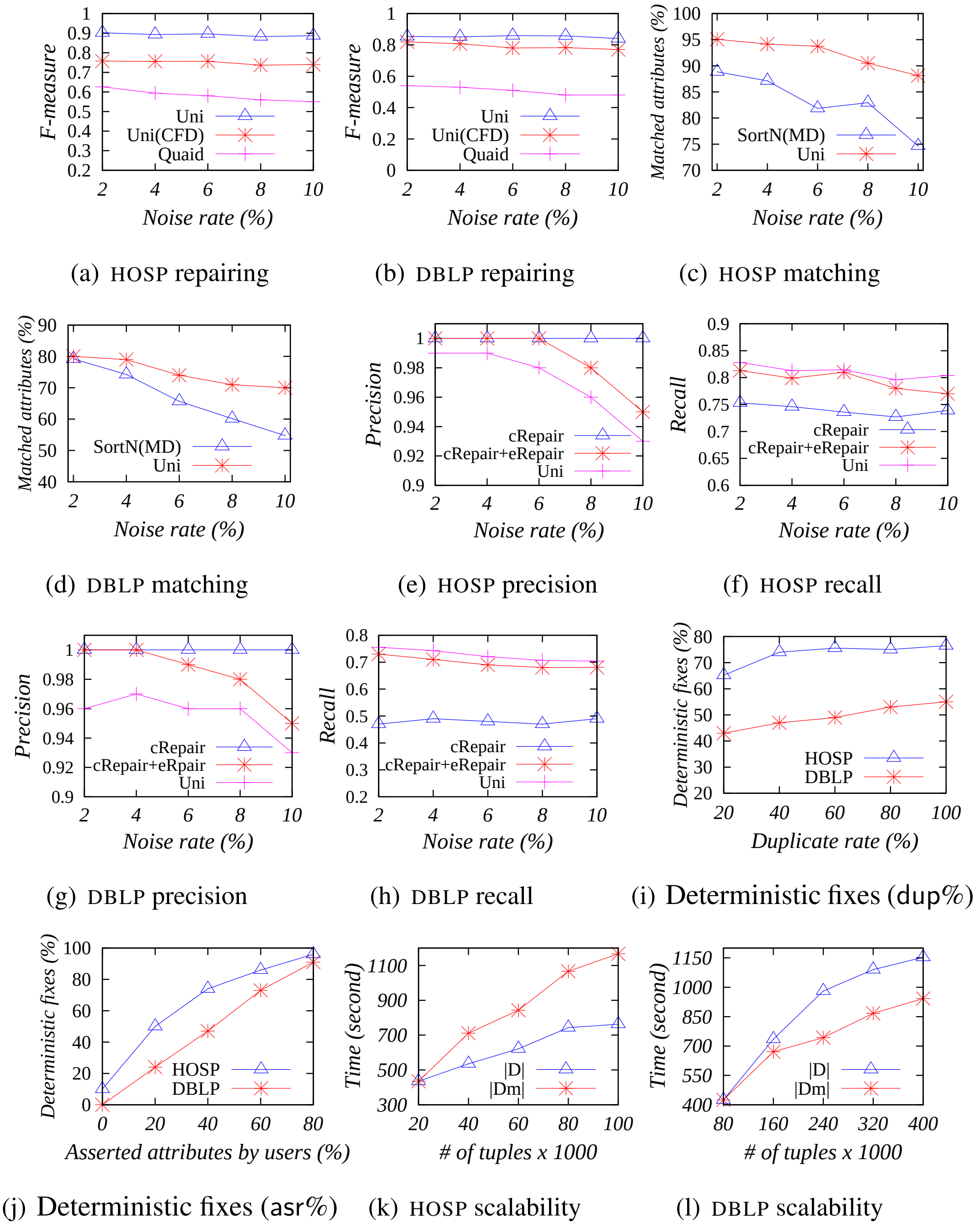
For record matching, (a) precision is the ratio of *true matches* (true positives) correctly found by an algorithm to all the duplicates found, and (b) recall is the ratio of true matches correctly found to all the matches between a dataset and master data. For data repairing, (a) precision is the ratio of attributes correctly updated to the number of all the attributes updated, and (b) recall is the ratio of attributes corrected to the number of all erroneous attributes.

All experiments were conducted on a Linux machine with a 3.0GHz Intel CPU and 4GB of Memory. Each experiment was run more than 5 times, and the average is reported here.

Experimental Results. We conducted five sets of experiments: (a) in the first two sets of experiments, we compared the effectiveness of our cleaning methods with both matching and repairing against its counterpart with only matching or only repairing; (b) we evaluated the accuracy of deterministic fixes, reliable fixes and possible fixes in the third set of experiments; (c) we evaluated the impact of the duplicate rate and asserted rate on the percentage of deterministic fixes found by our algorithm cRepair in the fourth set of experiments; and (d) the last set of experiments tested the scalability of Uni with both the size of dirty data and the size of master data. In all the experiments, we set the threshold for entropy and confidence to be 0.8 and 1.0, respectively. We used dirty datasets and master data consisting of 60K tuples each. We now report our findings.

Exp-1: Matching helps repairing. In the first set of experiments we show that matching indeed helps repairing. We compare the quality (F-measure) of fixes generated by Uni, Uni(CFD) and quaid. Fixing the duplicate rate dup% = 40%, we varied the noise rate noi% from 2% to 10%. Observe that dup% is only related to matching via MDs. To favor Uni(CFD) and quaid, which use CFDs only, we focused on the impact of various noise rates.

The results on HOSP data and DBLP data are reported in Figures 4.8(a) and 4.8(b), respectively, which tell us the following. (1) Uni clearly outperforms Uni(CFD) and quaid by up to 15% and 30%, respectively. This verifies that matching indeed helps repairing. (2) The F-measure decreases when noi% increases for all three approaches. However, Uni with matching is less sensitive to noi%, which is another benefit of unifying repairing with matching. (3) Even only with CFDs, our system Uni(CFD) still outperforms quaid, as expected. This is because quaid only generates possible fixes with heuristic, while Uni(CFD) finds both deterministic fixes and reliable fixes. This



(a) HOSP repairing (b) DBLP repairing (c) HOSP matching

(d) DBLP matching (e) HOSP precision (f) HOSP recall

(g) DBLP precision (h) DBLP recall (i) Deterministic fixes (dup%)

(j) Deterministic fixes (asr%) (k) HOSP scalability (l) DBLP scalability

Figure 4.8: Experimental results

also verifies that deterministic and reliable fixes are more accurate than possible fixes.

Exp-2: Repairing helps matching. In the second set of experiment, we show that repairing indeed helps matching. We evaluate the quality (F-measure) of matches found by (a) Uni and (b) SortN using MDs, denoted by SortN(MD). We used the same setting as in Exp-1. We also conducted experiments by varying the duplicate rate, but found that its impact is very small; hence we do not report it here.

The results are reported in Figures 4.8(c) and 4.8(d) for HOSP data and DBLP data, respectively. We find the following. (a) Uni outperforms SortN(MD) by up to 15%, verifying that repairing indeed helps matching. (b) The F-measure decreases when the noise rate increases for both approaches. However, Uni with repairing is less sensitive to noi%, which is consistent with our observation in the last experiments.

Exp-3: Accuracy of deterministic and reliable fixes. In this set of experiments we evaluate the accuracy (precision and recall) of (a) deterministic fixes generated in the first phase of UniClean, denoted by cRepair, (b) deterministic fixes and reliable fixes generated in the first two phases of UniClean, denoted by cRepair + eRepair, and (c) all fixes generated by Uni. Fixing dup% = 40%, we varied noi% from 2% to 10%. The results are reported in Figures 4.8(e)–4.8(h).

The results tell us the following: (a) Deterministic fixes have the highest precision, and are insensitive to the noise rate. However, their recall is low, since cRepair is "picky": it only generates fixes with asserted attributes. (b) Fixes generated by Uni have the lowest precision, but the highest recall, as expected. Further, their precision is quite sensitive to noi%. This is because the last step of UniClean is by heuristics, which generates possible fixes. (c) The precision and recall of deterministic fixes and reliable fixes by cRepair + eRepair are in the between, as expected. Further, their precision is also sensitive to noi%. From these we can see that the precision of reliable fixes and possible fixes is sensitive to noi%, but not their recall. Moreover, when noi% is less than 4%, their precision is rather indifferent to noi%.

Exp-4: Impact of dup% and asr% on deterministic fixes. In this set of experiments we evaluated the percentage of deterministic fixes found by algorithm cRepair.

Fixing the asserted rate asr% = 40%, we varied the duplicate rate dup% from 20% to 100%. Figure 4.8(i) shows the results. We find that the larger dup% is, the more deterministic fixes are found, as expected.

Fixing dup% = 40%, we varied asr% from 0% to 80%. The results are shown in Fig. 4.8(j), which tell us that the number of deterministic fixes found by cRepair highly depends on asr%. This is because to find deterministic fixes, cleaning rules are only applied to asserted attributes.

Exp-5: Scalability. The last experiments evaluated the scalability of Uni with the size $|D|$ of dirty data and the size $|D_m|$ of master data. We fixed noi% = 6% and dup% = 40% in these experiments. The results are reported in Figures 4.8(k) and 4.8(l) for HOSP and DBLP data, respectively.

Figure 4.8(k) shows two curves for HOSP data: one by fixing $|D_m|$ = 60K and varying $|D|$ from $20K$ to $100K$, and the other by fixing $|D|$ = 60K and varying $|D_m|$ from $20K$ to $100K$. The results show that Uni scales reasonably well with both $|D|$ and $|D_m|$. In fact Uni scales much better than quaid [CFG$^+$07]: quaid took more than 10 hours when $|D|$ is 80K, while it took Uni about 11 minutes. These results verify the effectiveness of our indexing structures and optimization techniques developed for Uni. The results are consistent for DBLP data, as shown in Fig. 4.8(l).

Summary. From the experimental results on real-life data, we find the following. (a) Data cleaning by unifying matching and repairing substantially improves the quality of fixes: it outperforms matching and repairing taken as independent processes by up to 30% and 15%, respectively. (b) Deterministic fixes and reliable fixes are highly accurate. For example, when the noise rate is no more than 4%, their precision is close to 100%. The precision decreases slowly when increasing noise rate. These tell us that it is feasible to find accurate fixes for real-life applications. (c) Candidate repairs generated by system UniClean are of high-quality: their precision is about 96%. (d) Our data cleaning methods scale reasonably well with the size of data and the size of master data. It is more than 50 times faster than quaid a data repairing tool using CFDs only.

Chapter 5

Inferring Data Currency and Consistency for Conflict Resolution

5.1 Introduction

In this chapter, we study a new approach for conflict resolution, by inferring both data currency and consistency (Chapter 5).

Given a database instance I_t with all tuples referring to the same real world entity, we want to identify a single tuple in which each attribute has *consistent* and *the most current value* (the latest value) taken from I_t, referred to as the *true values* of the entity *relative to* I_t. The need for studying this problem is evident in data integration, where conflicts are often generated by values that refer to the same real world entity but come from multiple data sources and time.

It is also common to find multiple values of the same entity presenting in one database. While these values were *once correct*, *i.e.,* they were the true values of the entity at some time, some of them may have become *out of date* and thus *inconsistent*. The need for resolving conflicts for, *e.g.,* data fusion [BN08, DN09], data cleaning [ABC03] and query answering with current values [FGW11] becomes evident.

However, it is already highly nontrivial to find consistent values for an entity [ABC03, CFG$^+$07]. Moreover, it is hard to identify the most current values of one entity [FGW11] since in the real world, reliable timestamps are often absent or unreliable [ZDI10, RG95]. Add to this the complication that one has to find the entity values that are *both* consistent *and* most current to resolve conflicts.

Example 5.1.1: The photo in Fig. 5.1 is known as "V-J Day in Times Square".

Figure 5.1: V-J Day

The nurse and sailor in the photo have been identified as Edith Shain and George Mendonça, respectively, and their information is collected in sets E_1 and E_2 of tuples, respectively, shown in Fig. 5.2.

We want to find the true values of these entities, *i.e.,* a tuple t_1 for Edith (resp. a tuple t_2 for George) such that the tuple has the most current and consistent attribute values for her (resp. his) status, job, the number of kids, city, AC (area code), zip and county in E_1 (resp. E_2). However, the values in E_1 (E_2) have conflicts, and worse still, they do not carry timestamps. They do not tell us, for instance, whether Edith still lives in NY, or even whether she is still alive. □

Although the situation is bad, it is not hopeless. We can often discover certain currency orders based on the semantics of the data. Also, dependencies such as CFDs have shown effective in improving the consistency of the data. Better still, data currency and consistency interact with each other. When they are taken together, we can often find some true values from inconsistent tuples, even when without timestamps, as shown

		name	status	job	kids	city	AC	zip	county
E_1	r_1:	*Edith Shain*	*working*	*nurse*	*0*	*NY*	*212*	*10036*	*Manhattan*
	r_2:	*Edith Shain*	*retired*	*n/a*	*3*	*SFC*	*415*	*94924*	*Dogtown*
	r_3:	*Edith Shain*	*deceased*	*n/a*	null	*LA*	*213*	*90058*	*Vermont*
E_2	r_4:	*George Mendonça*	*working*	*sailor*	*0*	*Newport*	*401*	*02840*	*Rhode Island*
	r_5:	*George Mendonça*	*retired*	*veteran*	*2*	*NY*	*212*	*12404*	*Accord*
	r_6:	*George Mendonça*	*unemployed*	*n/a*	*2*	*Chicago*	*312*	*60653*	*Bronzeville*

Figure 5.2: Instances E_1 for entity Edith and E_2 for George

Currency constraints:	φ_1: $\forall t_1, t_2$ (t_1[status] = “working” $\wedge$ t_2[status] = “retired” $\rightarrow t_1 \prec_{\text{status}} t_2$) φ_2: $\forall t_1, t_2$ (t_1[status] = “retired” $\wedge$ t_2[status] = “deceased” $\rightarrow t_1 \prec_{\text{status}} t_2$) φ_3: $\forall t_1, t_2$ (t_1[job] = “sailor” $\wedge$ t_2[job] = “veteran” $\rightarrow t_1 \prec_{\text{job}} t_2$) φ_4: $\forall t_1, t_2$ ($t_1[\text{kids}] < t_2[\text{kids}] \rightarrow t_1 \prec_{\text{kids}} t_2$) φ_5: $\forall t_1, t_2$ ($t_1 \prec_{\text{status}} t_2 \rightarrow t_1 \prec_{\text{job}} t_2$) φ_6: $\forall t_1, t_2$ ($t_1 \prec_{\text{status}} t_2 \rightarrow t_1 \prec_{\text{AC}} t_2$) φ_7: $\forall t_1, t_2$ ($t_1 \prec_{\text{status}} t_2 \rightarrow t_1 \prec_{\text{zip}} t_2$) φ_8: $\forall t_1, t_2$ ($t_1 \prec_{\text{city}} t_2 \wedge t_1 \prec_{\text{zip}} t_2 \rightarrow t_1 \prec_{\text{county}} t_2$)
Constant CFDs:	ψ_1 : (AC = *213* $\rightarrow$ city = *LA*); ψ_2 : (AC = *212* $\rightarrow$ city = *NY*);

Figure 5.3: Currency constraints and constant CFDs

below.

Example 5.1.2: From the semantics of the data, we can deduce the *currency constraints* and CFDs shown in Fig. 5.3.

(1) Currency constraints. We know that for each person, status only changes from *working* to *retired* and from *retired* to *deceased*, but not from *deceased* to *working* or *retired*. These can be expressed as φ_1 and φ_2 given in Fig. 5.3, referred to as *currency constraints*. Here $t_1 \prec_{\text{status}} t_2$ denotes a partial currency order defined on the attribute status, indicating that t_2 is *more current* than t_1 *in attribute* status. Similarly, we know that job can only change from *sailor* to *veteran* but not the other way around. We can express this as currency constraint φ_3, shown in Fig. 5.3. Moreover, the number of kids typically increases monotonically. We can express this as φ_4, assuring that t_2 is more current than t_1 in attribute kids if $t_1[\text{kids}] < t_2[\text{kids}]$.

In addition, we know that for each person, if tuple t_2 is more current than t_1 in attribute status, then t_2 is also more current than t_1 in job, AC and zip. Furthermore, if t_2 is more current than t_1 in attributes city and zip, it also has a more current county than t_1. These can be expressed as currency constraints φ_5–φ_8.

(2) *Constant CFDs.* In the US, if the AC is *213* (resp. *212*), then the city must be *LA* (resp. *NY*). These are expressed as conditional functional dependencies ψ_1 and ψ_2 shown in Fig. 5.3.

We can apply these constraints to the set E_1 of tuples given in Fig. 5.2, to improve the currency and consistency of the data. By *interleaving* inferences of data currency and data consistency, we can actually identify the true values of entity Edith, as follows:

(a) from the currency constraints φ_1 and φ_2, we can conclude that her latest status is *deceased*;

(b) similarly, by φ_4, we find that her true kids value is *3* (assuming null $< k$ for any number k);

(c) from (a) above and φ_5–φ_7, we know that her latest job, AC and zip are *n/a*, *213* and *90058*, respectively;

(d) after currency inferences (a) and (c), we can apply the CFD ψ_1 and find her latest city as *LA*; and

(e) after the consistency inference (d), from (c) and (d) we get her latest county as *Vermont*, by applying the currency constraint φ_8.

Now we have identified a single tuple

t_1 = (Edith Shain, deceased, n/a, 3, LA, 213, 90085, Vermont)

as the true values of Edith Shain relative to the set E_1 of tuples (the address is for her cemetery). □

This example shows that data currency and consistency could be interleaved when resolving conflicts. In addition, both data currency and data consistency can be specified with constraints, and hence, can be processed in a uniform logical framework.

While the need for deducing the consistent and most current values has been promoted for conflict resolution [DN09, MA06], prior work mostly assumes the availability of timestamps. Previous work on data quality focuses on either data consistency (*e.g.,* [ABC03, FGJK08, CFG$^+$07, YENO10]) or data currency (*e.g.,* [FGW11]). However, no models or algorithms are yet in place to combine data consistency and currency for *conflict resolution*.

We contend that this work provides fundamental results for conflict resolution, and proposes a practical solution by inferring data currency and data consistency.

Related work. Conflict resolution has been studied for decades, started from [Day83]. It aims to combine data from different sources into a single representation (see [BN08, DN09] for surveys). In that context, inconsistencies are typically resolved by selecting the max, min, avg, any value [BN08]. While the need for data currency was also observed there (*e.g.,* [DN09, MA06]), previous work identifies current values only by using *timestamps*. This work differs from the traditional work in the following. (1) We revise the conflict resolution problem to identify values of entities that are both *consistent* and most *current*. (2) We *do not* assume the availability of timestamps, which are often missing in practice [ZDI10, RG95]. (3) We resolve conflicts by using currency constraints and CFDs [ABC03, FGJK08, CFG$^+$07], instead of picking max, min, avg or any value. (4) We employ *automated reasoning* to identify true values by unifying the inferences of data currency and consistency.

There has been work on truth discovery from data sources [DBES09b, GAMS10, YHY08]. Their approaches include (1) vote counting and probabilistic computation based on the trustworthiness of data sources [GAMS10, YHY08]; (2) source dependencies to find copy relationships and reliable sources [DBES09b]; and (3) employing lineage information and probabilities [Wid05]. In contrast, we assume no information about the accuracy of data sources, but derive true values based on data currency and consistency. In addition, we adopt a logical approach via automated reasoning about constraints, as opposed to probabilistic computation. This work is complementary to the previous work and can be combined with the prior approaches.

This work extends [FGW11, FGJK08]. A data currency model was presented in [FGW11] with partial currency orders and denial constraints [ABC03]. CFDs were studied for specifying data consistency [FGJK08]. This work differs from [FGW11, FGJK08] in the following. (1) We propose a conflict resolution model that combines data currency and consistency. In contrast, [FGW11] only studies data currency, while [FGJK08] only considers data consistency. (2) We *interleave* inferences of data currency and consistency, which is far more intriguing than handling currency and consistency separately, and requires new techniques to capture the interaction between the two. (3) We use currency constraints, which are simpler than denial constraints, to strike a balance between the complexity of inferring true values and the expressivity needed for specifying currency (Section 5.4). (4) *No practical algorithms* were given in [FGW11] for deriving current values.

Previous work on data consistency [ABC03, FGJK08, CFG$^+$07, YENO10, GSTZ03] has been focusing on consistent query answering and data repairing [Ber11], topics different from conflict resolution. The study of preferred repairs [GSTZ03] also advocates partial orders. It differs from the currency orders we study here in that they use PTIME functions to rank different repairs over the entire database, whereas we derive the currency orders by automated *reasoning about both* available partial temporal information and currency constraints. Preferred repairs are implemented by [CFG$^+$07] via a cost metric, and by [YENO10] based on a decision theory, which can be incorporated into our framework.

There has been a large body of work on temporal databases (see [CT05] for a survey). In contrast to that line of work, we do not assume the availability of timestamps. It has also recently been shown that temporal information helps record linkage identify records that refer to the same entity [LDMS11]. Here we show that data currency helps conflict resolution as well, a different process that takes place *after* record linkage has identified tuples pertaining to the same entity. While [LDMS11] is based on timestamps, we do not assume it here.

Organization. The rest of the chapter is organized as follows. We propose a model for specifying conflicts in Section 5.2, based on data currency and consistency, and introduce a framework for resolving conflicts in Section 5.3. Problems fundamental to conflict resolution are studied in Section 5.4, and practical algorithms underlying the conflict resolution framework are developed in Section 5.5. An experimental study is reported in Section 5.6

5.2 A Conflict Resolution Model

We first show how to capture conflicts in terms of data currency and consistency. We start with currency (Section 5.2.1) and consistency (Section 5.2.2) specifications. We then present the model (Section 5.2.3).

5.2.1 Data Currency

We specify the currency of data by means of (a) partial currency orders, and (b) currency constraints.

Data with partial currency orders. Consider a relation schema $R = (A_1, \ldots, A_n)$, where each attribute A_i has a domain $\mathsf{dom}(A_i)$. In this work we focus on *entity instances* I_e of R, which are sets of tuples of R *all* pertaining to the *same* real-world entity e, and are typically much smaller than a database instance. Such entity instances can be identified by *e.g.,* record linkage techniques (see [EIV07] for a survey).

For an attribute $A_i \in R$ and an entity instance I_e of R, we denote by $\mathsf{adom}(I_e.A_i)$ the set of A_i-attribute values that occur in I_e, referred to as *the active domain of A_i in I_e.*

For example, two entity instances are given in Fig. 5.2: $E_1 = \{r_1, r_2, r_3\}$ for entity “Edith”, and $E_2 = \{r_4, r_5, r_6\}$ for “George”; and $\mathsf{adom}(E_1.\mathsf{city}) = \{NY, SFC, LA\}$.

A *temporal instance I_t* of I_e is given as $(I_e, \preccurlyeq_{A_1}, \ldots, \preccurlyeq_{A_n})$, where each $\preccurlyeq_{A_i}$ is a partial order on I_e, referred to as the *currency order for attribute A_i* for the entity represented by I_e. For $t_1, t_2 \in I_e$, $t_1 \preccurlyeq_{A_i} t_2$ if and only if (iff) either t_1 and t_2 share the same A_i-attribute value (*i.e.,* $t_1[A_i] = t_2[A_i]$), or that $t_2[A_i]$ is more current than $t_1[A_i]$ (denoted by $t_1 \prec_{A_i} t_2$).

Intuitively, currency orders represent *available* temporal information about the data. Observe that $\preccurlyeq_{A_i}$ is a *partial order*, possibly empty. For example, for E_1 above, we only know that $r_3 \preccurlyeq_{\mathsf{kids}} r_1$ and $r_3 \preccurlyeq_{\mathsf{kids}} r_2$ since $r_3[\mathsf{kids}]$ is null, which are in the currency order $\preccurlyeq_{\mathsf{kids}}$, while the currency orders for other attributes are empty, excluding the case when tuples carry the same attribute value. Similarly for E_2. In particular, $t_1 \preccurlyeq_{A_i} t_2$ if $t_1[A_i]$ is null, *i.e.,* an attribute with value missing is ranked the lowest in the currency order.

Current instances. Currency orders are often incomplete. Hence we consider possible completions of currency orders.

A *completion I_t^c* of I_t is a temporal instance $I_t^c = (I_e, \preccurlyeq_{A_1}^c, \ldots, \preccurlyeq_{A_n}^c)$, such that for each $i \in [1, n]$, (1) $\preccurlyeq_{A_i} \subseteq \preccurlyeq_{A_i}^c$, and (2) for all tuples $t_1, t_2 \in I_e$, either $t_1 \preccurlyeq_{A_i}^c t_2$ or $t_2 \preccurlyeq_{A_i}^c t_1$. That is, $\preccurlyeq_{A_i}^c$ induces a *total order* on tuples in I_e.

That is, I_t^c totally sorts the attribute values in I_e such that the most current value of each attribute is the last in the order.

We define *the most current A_i-attribute value of I_t^c* to be $t[A_i]$ that comes last in the total order $\preccurlyeq_{A_i}^c$. The *current tuple* of I_t^c, denoted by $\mathsf{LST}(I_t^c)$ (*i.e.,* last), is the tuple t_l such that for each attribute A_i, $t_l[A_i]$ is the most current A_i-value of I_t^c, *i.e.,* t_l contains the most current values from I_t^c.

Currency constraints. One can derive additional currency information from the semantics of the data, which is modeled as *currency constraints.* A currency constraint φ is of the form

$$\forall t_1, t_2\ (\omega \rightarrow t_1 \prec_{A_r} t_2),$$

where ω is a conjunction of predicates of the form: (1) $t_1 \prec_{A_l} t_2$, *i.e.,* t_2 is more current than t_1 in attribute A_l; (2) $t_1[A_l]$ op $t_2[A_l]$, where op is one of $=,\neq,>,<,\leq,\geq$; and (3) $t_i[A_l]$ op c for $i \in \{1,2\}$, where c is a constant.

In contrast to denial constraints in the model of [FGW11], currency constraints are defined on two tuples, like functional dependencies. Such constraints suffice to specify currency information commonly found in practice (see, *e.g.,* Example 5.1.2).

Currency constraints are interpreted over completions I_t^c of I_t. We say that I_t^c *satisfies* φ, denoted by $I_t^c \models \varphi$, if for any two tuples t_1, t_2 in I_e, if these tuples and related order information in I_t^c satisfy the predicates in ω, following the standard semantics of first-order logic, then $t_1 \prec_{A_r}^c t_2$.

We say that I_t^c satisfies a set Σ of currency constraints, denoted by $I_t^c \models \Sigma$, if $I_t^c \models \varphi$ for all $\varphi \in \Sigma$.

Example 5.2.1: Recall the entity instances E_1 and E_2 given in Fig. 5.2. Currency constraints on these instances include φ_1–φ_8 as specified in Fig. 5.3 and interpreted in Example 5.1.2.

It is readily verified that for any completion E_1^c of E_1, if it satisfies these constraints, it yields $\mathsf{LST}(E_1^c)$ of the form (*Edith*, *deceased*, *n/a*, *3*, x_{city}, *213*, *90058*, x_{county}) for Edith, in which the most current values for attributes name, status, job, kids, AC and zip are deduced from the constraints and remain unchanged, while x_{city} and x_{county} are values determined by the total currency order given in E_1^c. Observe that the values of the current tuple are taken from *different tuples* in E_1, *e.g.,* kids = *3* from r_2 and AC = *213* from r_3.

Similarly, for any completion of E_2, its current tuple has the form (*George*, x_{status}, x_{job}, *2*, $x_{\mathsf{city}}, x_{\mathsf{AC}}, x_{\mathsf{zip}}, x_{\mathsf{county}}$), if they satisfy all constraints. Hence, currency constraints help us find some but not *all* of the most current values of entities. □

5.2.2 Data Consistency

To specify the consistency of data, we use *constant* CFDs [FGJK08] (as depicted in Chater 2).

A constant CFD ψ on a relation schema R is of the form $t_p[X] \to t_p[B]$, where (1) $X \subseteq R$, $B \in R$; and (2) t_p is the *pattern tuple* of ψ with attributes in X and B, where for each A in $X \cup \{B\}$, $t_p[A]$ is a *constant* in $\mathsf{dom}(A)$ of A.

For example, ψ_1 and ψ_2 in Table 5.3 are constant CFDs on the relation of Table 5.2, as interpreted in Example 5.1.2.

Such CFDs are defined on *the current tuple* of a completion. Consider a completion

I_t^c of I_t and let $t_l = \mathsf{LST}(I_t^c)$ be the current tuple of I_t^c. We say that the completion I_t^c *satisfies* a constant CFD $\psi = t_p[X] \rightarrow t_p[B]$, denoted by $I_t^c \models \psi$, iff when $t_l[X] = t_p[X]$ then $t_l[B] = t_p[B]$.

Intuitively, this assures that if $t_l[X] = t_p[X]$ and if $t_l[X]$ contains the most current X-attribute values, then $t_l[B]$ can be *repaired* by taking the value $t_p[B]$ in the pattern, and moreover, $t_l[B]$ is the most current value in attribute B.

We say that I_t^c *satisfies* a set Γ of constant CFDs, denoted as $I_t^c \models \Gamma$, iff $I_t^c \models \psi$ for each $\psi \in \Gamma$.

Observe that a constant CFD is defined on a *single tuple* $\mathsf{LST}(I_t^c)$. In light of this, we do not need general CFDs of [FGJK08] here, which are typically defined on *two tuples*.

Example 5.2.2: Recall the current tuples for E_1 in Example 5.2.1. Then all completions of E_1 that satisfy ψ_1 in Fig. 5.3 have the form (*Edith*, *deceased*, *n/a*, *3*, *LA*, *213*, *90058*, *Vermont*), in which x_{city} is instantiated as *LA* by ψ_1, and as a result, x_{county} becomes *Vermont* by the currency constraint φ_8. □

5.2.3 Conflict Resolution

We are ready to bring currency and consistency together.

Specifications. A *specification* $S_e = (I_t, \Sigma, \Gamma)$ *of an entity* consists of (1) a temporal instance $I_t = (I_e, \preccurlyeq_{A_1}, \ldots, \preccurlyeq_{A_n})$; (2) a set Σ of currency constraints; and (3) a set Γ of constant CFDs. A completion $I_t^c = (I_e, \preccurlyeq_{A_1}^c, \ldots, \preccurlyeq_{A_n}^c)$ of I_t is a *valid completion* of S_e if I_t^c satisfies both Σ and Γ. We say that S_e is *valid* if there exists a valid completion I_t^c of S_e, *e.g.,* the specification of E_1 (or E_2) and the constraints in Fig. 5.3 is valid.

True values. There may be many valid completions I_t^c, each leading to a possibly different current tuple $\mathsf{LST}(I_t^c)$. When two current tuples differ in some attribute, there is *a conflict*. We aim to resolve such conflicts. If all such current tuples agree on *all* attributes, then the specification is conflict-free, and a *unique* current tuple exists for the entity e specified by S_e. In this case, we say that this tuple is the true value of e.

More formally, the *true value* of S_e, denoted by $\mathsf{T}(S_e)$, is the *single* tuple t_c such that for *all valid* completions I^c of S_e, $t_c = \mathsf{LST}(S_e)$, if it exists. For each attribute A_i of R, we call $t_c[A_i]$ the *true value* of A_i in S_e.

The conflict resolution problem. Consider a specification $S_e = (I_t, \Sigma, \Gamma)$, where $I_t = (I_e, \preccurlyeq_{A_1}, \ldots, \preccurlyeq_{A_n})$. Given S_e, conflict resolution is to find the minimum amount of additional currency information such that the true value exists.

The additional currency information is specified in terms of a *partial temporal order* $O_t = (I, \preccurlyeq'_{A_1}, \ldots, \preccurlyeq'_{A_n})$. We use $S_e \oplus O_t$ to denote the extension $S'_e = (I'_t, \Sigma, \Gamma)$ of S_e by enriching I_t with O_t, where $I'_t = (I_e \cup I, \preccurlyeq_{A_1} \cup \preccurlyeq'_{A_1}, \ldots, \preccurlyeq_{A_n} \cup \preccurlyeq'_{A_n})$. We only consider partial temporal orders O_t such that $\preccurlyeq_{A_i} \cup \preccurlyeq'_{A_i}$ is a partial order for all $i \in [1,n]$.

We use $|O_t|$ to denote $\Sigma_{i \in [1,n]} |\preccurlyeq'_{A_i}|$, *i.e.,* the sum of the sizes of all the partial orders in O_t.

Given a valid specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity, the *conflict resolution problem* is to find a partial temporal order O_t such that (a) $\mathsf{T}(S_e \oplus O_t)$ exists and (b) $|O_t|$ is minimum.

Example 5.2.3: Recall from Example 5.2.1 the current tuples for George. Except for name and kids, we do not have a unique current value for the other attributes. Nonetheless, if a partial temporal order O_t with, *e.g.,* $r_6 \prec_{\mathsf{status}} r_5$ is provided by the users (*i.e.,* status changes from *unemployed* to *retired*), then the true value of George in E_2 can be derived as (*George*, *retired*, *veteran*, *2*, *NY*, *212*, *12404*, *Accord*) from the currency constraints and CFDs of Fig. 5.3. □

5.3 A Conflict Resolution Framework

We propose a framework for conflict resolution. As depicted in Fig. 5.4, given a specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity e, the framework is to find the true value $\mathsf{T}(S_e)$ of e by reasoning about data currency and consistency, and by interacting with the users to solicit additional data currency information.

The framework provides the users with suggestions. A *suggestion* is a minimum set $\mathcal{A}$ of attributes of e such that if the true values of these attributes are provided by the users, $\mathsf{T}(S_e)$ is automatically deduced from the users' input, Σ, Γ and I_t. The true values for $\mathcal{A}$ are represented as a temporal order O_t. More specifically, the framework deduces $\mathsf{T}(S_e)$ as follows.

(1) *Validity checking.* It first inspects whether $S_e \oplus O_t$ is valid, via automated reasoning, where O_t is a partial temporal order provided by the users, *initially empty* (see step (4) below for details about O_t). If so, it follows the 'Yes' branch. Otherwise the users need to revise O_t by following the 'No' branch.

(2) *True value deducing.* After $S_e \oplus O_t$ is validated, it derives as many true values for the attributes of e as possible, via automated reasoning.

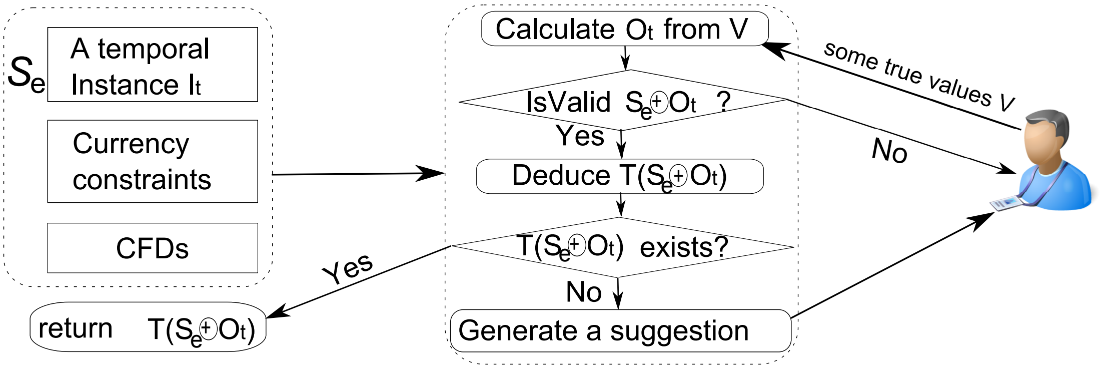

S_e
A temporal Instance I_t
Currency constraints
CFDs
Calculate O_t from V
IsValid $S_e \oplus O_t$?
Yes
Deduce $T(S_e \oplus O_t)$
$T(S_e \oplus O_t)$ exists?
No
Generate a suggestion
Yes
return $T(S_e \oplus O_t)$
some true values V
No

Figure 5.4: Framework overview

(3) *Finding the true value*. If $\mathsf{T}(S_e \oplus O_t)$ exists, it computes and returns it following the 'Yes' branch. Otherwise, it follows the 'No' branch and goes to step (4).

(4) *Generating suggestions*. It computes a suggestion $\mathcal{A}$ along with its candidate values taken from the active domain of S_e, such that if the users pick and validate the true values for $\mathcal{A}$, then $\mathsf{T}(S_e \oplus O_t)$ is warranted to be found. The users are expected to provide V, the true values of *some attributes* in $\mathcal{A}$, represented as a partial temporal order O_t. Given O_t, $S_e \oplus O_t$ is constructed and the process goes back to step (1).

The process proceeds until $\mathsf{T}(S_e \oplus O_t)$ is found, or when the users opt to settle with true values for a subset of attributes of e. That is, if users do not have sufficient knowledge about the entity, they may let the system derive true values for as many attributes as possible, and revert to the traditional methods to pick the max, min, avg, any values for the rest of the attributes.

Remarks. (1) To specify users' input, let I_t in S_e be $(I_e, \preccurlyeq_{A_1}, \ldots, \preccurlyeq_{A_n})$ and $\mathcal{A} \cup \mathcal{A}' \cup \mathcal{B}$ = $\{A_1, \ldots, A_n\}$, where (i) $\mathcal{A}$ is the set of attributes identified in step (4) for which the true values are unknown; (ii) for $\mathcal{B}$, their true values $\mathsf{V}_{\mathcal{B}}$ have been deduced (step (2)); and (iii) $\mathcal{A}'$ is the set of attributes whose true values can be deduced from $\mathsf{V}_{\mathcal{B}}$ and the suggestion for $\mathcal{A}$. Given a suggestion, the user is expected to provide a set V of true values for (a subset of) $\mathcal{A}$ that they are confident of. Here V consists of either the candidate values taken from the suggestion, or some *new* values not in the active domains of S_e that users opt to choose. The users *do not* have to enter values for *all* attributes in $\mathcal{A}$.

From the input V, a partial temporal order O_t is automatically derived, by treating V as the most current values of those attributes involved. Indeed, O_t has the form $(I_e \cup \{t_o\}, \preccurlyeq'_{A_1}, \ldots, \preccurlyeq'_{A_n})$, where t_o is a new tuple such that for all attributes A, $t_o[A] =$

$\mathsf{V}(A)$ if V has a value $\mathsf{V}(A)$ for A, and $t_o[A] = \mathsf{null}$ otherwise, while $t_o[\mathcal{B}] = \mathsf{V}_{\mathcal{B}}$ remains unchanged. Moreover, $\preccurlyeq'_A$ extends $\preccurlyeq_A$ by including $t[A] \preccurlyeq_A t_o[A]$ if $t_o[A] \neq \mathsf{null}$, for all tuples $t \in I_e$. Then $S_e \oplus O_t$ can be readily defined.

(2) There have been efficient methods for discovering constant CFDs, *e.g.,* [FGLX11]. Along the same lines as CFD discovery [CM08, FGLX11], automated methods can be developed for discovering currency constraints from (possibly dirty) data. With certain quality metric in place [CM08], the constraints discovered can be as accurate as those manually designed (such as those given in Fig. 5.3), and can be used by the framework as input.

(3) To simplify the discussion we do not allow users to change constraints in S_e.

(4) We assume that the values in entity instances were once correct. When a temporal instance contains errors, one may inspect different samples and only take those currency orders that are consistent among the samples. or have sufficient support (*e.g.,* frequency).

5.4 Fundamental Problems

In this section, we identify fundamental problems associated with conflict resolution based on both data currency and consistency, and establish their complexity. These results are not only of theoretical interest, but also tell us where the complexity arises, and hence guide us to develop effective (heuristic) algorithms.

Satisfiability. The first one is the *satisfiability problem* for entity specifications. It is to decide, given a specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity, whether S_e is *valid*, *i.e.,* whether there exists a valid completion of S_e.

Intuitively, it is to check whether S_e makes sense, *i.e.,* whether the currency constraints, constant CFDs and partial orders in S_e, when put together, have conflicts themselves. The analysis is needed by the step (1) of the framework of Fig. 5.4. In practice, this analysis tells us whether we have to revise constraints in S_e, or ask users to validate its partial orders.

The problem is obviously important, but is NP-complete. One might think that the absence of currency constraints or CFDs would simplify the analysis. Unfortunately, its intractability is rather *robust.*

Theorem 5.4.1: *The satisfiability problem for entity specifications is NP-complete. It* *remains NP-hard for valid specifications* $S_e = (I_t, \Sigma, \Gamma)$ *of an entity when (1) both* Σ *and* Γ *are fixed; (2)* $\Gamma = \emptyset$*,* i.e., *when only currency constraints are present; or (3)* $\Sigma = \emptyset$*,* i.e., *when only constant CFDs are present.* □

Proof. [*] For the upper bound it suffices to observe that the following NP algorithm correctly decides whether a given specification has a valid completion. Given a specification $S_e = (I_t, \Sigma, \Gamma)$, the algorithm simply guesses a completion I_t^c of I_t and then checks whether (i) $I_t^c \models \Sigma$; and (ii) $I_t^c \models \Gamma$. If the guessed completion passes these checks, then the algorithm returns "yes". Otherwise, the guessed completion is rejected. Note that a "guess" simply completes the partial orders on the values of I_t, and there are finitely many guesses in total. The algorithm is in NP since checking can be done in PTIME.

The NP-lower bound is established by reduction from the 3-satisfiability problem. An instance of the 3-satisfiability problem is formula $\varphi = C_1 \wedge \cdots \wedge C_r$ with $C_j = \ell_1^j \vee \ell_2^j \vee \ell_3^j$, where for $k \in \{1,2,3\}$ and $j \in [1,r]$, ℓ_k^j is either a variable or a negation of a variable from a set $X = \{x_1, \ldots, x_n\}$ of variables. It is to determine whether φ is satisfiable, *i.e.,* whether there exists a truth assignment of variables in X that satisfies φ. This problem is known to be NP-complete (cf. [Pap94]).

Given φ, we define a specification $S_e = (I_t, \Sigma, \Gamma)$ such that there exists a valid completion of S_e iff φ is satisfiable. The specification S_e consists of a temporal instance I_t of schema $R(D, C, P, U, V, W)$ and a fixed set of currency constraints Σ. No constant CFDs are defined in S_e. Intuitively, D is to distinguish between tuples that encode truth assignments and tuples that correspond to clauses in φ; C is to identify variables (by x_i) and clauses (by $j \in [1,r]$); P is used to enforce the validity of clauses and finally, U, V and W represent the positions (1, 2 and 3, resp) of variables in each clause.

We first explain how the temporal instance I_t of R together with the currency constraints in Σ is to encode truth assignments for X and clauses in φ. More specifically, for each variable $x_i \in X$, we use two constants a_i and b_i such that $a_i \preccurlyeq_A b_i$ encodes that x_i is set to true, whereas $b_i \preccurlyeq_A a_i$ encodes that $\bar{x}_i$ is set to true (or, equivalently that x_i is set to false). Here A ranges over attributes U, V and W. More specifically, for each variable $x_i \in X$ we include two tuples in I_t:

$$(0, x_i, 0, a_i, a_i, a_i) \quad \text{and} \quad (0, x_i, 0, b_i, b_i, b_i).$$

These encode truth assignments of X. To ensure that the choice of truth value for

[*] This proof is a joint work with Floris Geerts.

variables is consistent, we include the following currency constraints in Σ:

$$\begin{aligned}\forall t_1, t_2 \in R(t_1[D] = 0 \wedge t_2[D] = 0 \wedge t_1[C] = t_2[C] \wedge \\ t_1[A] \prec t_2[A] \rightarrow t_1[B] \prec t_2[B]),\end{aligned}$$

where A and B range over distinct pairs taken form $\{U, V, W\}$. These currency constraints enforce that variables x_i are set to true (resp. false) independent of the position at which they appear in clauses (*i.e.,* in attribute U, V or W).

We next consider the clauses in φ. Let $C_j = \ell_1^j \vee \ell_2^j \vee \ell_3^j$. observe that this can be equivalently written as $\bar{\ell}_1^j \wedge \bar{\ell}_2^j \rightarrow \ell_3^j$. For instance, consider a clause $C = x_1 \vee \bar{x}_2 \vee \bar{x}_3$. This is equivalent to $\bar{x}_1 \wedge x_2 \rightarrow \bar{x}_3$. Given this, we include two tuples in I_t for each clause:

$$(1, j, 1, v_1, v_2, v_3) \quad \text{and} \quad (1, j, 2, v_1', v_2', v_3'),$$

where $v_i = a_k$ and $v_i' = b_k$ if $\bar{\ell}_i^j = x_k$, and $v_i = b_k$ and $v_i' = a_k$ if $\ell_i^j = x_k$, for $i = 1, 2$, and conversely for $i = 3$. The example clause C is thus encoded by $(1, \_, 1, b_1, a_2, b_3)$ and $(1, \_, 2, a_1, b_2, a_3)$. The connection between truth assignments selected by completions and the validity of clauses is established by means of the following currency constraint:

$$\begin{aligned}\forall t_1, t_2 \in R(t_1[D] = 1 \wedge t_2[D] = 1 \wedge t_1[C] = t_2[C] \wedge \\ t_1[P] = 1 \wedge t_1[P] = 2 \wedge \\ t_1[U] \prec t_2[U] \wedge t_1[V] \prec t_2[V] \rightarrow t_1[W] \prec t_2[W]).\end{aligned}$$

This constraint tells us that whenever the truth assignment (represented by a completion) makes $\bar{\ell}_1^j \wedge \bar{\ell}_2^j$ true, then it must also make ℓ_3^j true.

We next show the correctness of the reduction. Suppose that φ is true and let μ_X be a satisfying truth assignment. We define a valid completion of S_e as follows: For attributes D, C and P we order the tuples in I_t arbitrarily. For attributes U (and consequently also for V and W by the currency constraints) we set $a_i \preccurlyeq_U^c b_i$ if $\mu_X(x_i)$ is true, and $b_i \preccurlyeq_U^c a_i$ otherwise. We need to verify that the second currency constraint is satisfied. This follows immediately from the fact that each clause is satisfied by μ_X. Conversely, suppose that we have a valid completion of S_e. From this, we define μ_X by simply setting $\mu_X(x_i) = 1$ if $a_i \preccurlyeq_U^c b_i$ and $\mu_X(x_i) = 0$ otherwise. Similarly as above, it is readily verified that μ_X satisfies all the clauses. Indeed, this follows from the second currency constraint given above.

It remains to show that the satisfiability problem is NP-complete when (a) Σ and Γ are fixed; (b) $\Gamma = \emptyset$; or (c) $\Sigma = \emptyset$. Since we have shown that the satisfiability problem is in NP, for general Σ and Γ, it suffices to show the lower bounds. Furthermore, observe the proof above uses (i) a fixed set of currency constraints, *i.e.,* the currency constraints Σ are independent of the input instance φ, and (ii) it does not use any constant CFDs. In other words, (a) and (b) follow directly from the lower bound proof given above. It remains to show (c), *i.e.,* the satisfiability problem is NP-hard even when only constant CFDs are present.

We establish this lower bound by reduction from the complement of the tautology problem, which is known to coNP-complete (cf. [Pap94]). An instance of the tautology problem is a formula $\varphi = C_1 \vee \cdots \vee C_r$, where $C_j = \ell_1^j \wedge \ell_2^j \wedge \ell_3^j$ and each ℓ_k^j is either a variable or a complement of a variable from $X = \{x_1, \ldots, x_n\}$. It is to determine whether φ is true for all truth assignments of X. We define a specification $S_e = (I_t, \Sigma = \emptyset, \Gamma)$ such that S_e has a valid completion iff φ is not a tautology.

The temporal instance I_t of S_e is an instance of schema $R'(X_1, \ldots, X_n, C)$; it consists of two tuples $(0,0,\ldots,0)$ and $(1,1,\ldots,1)$. We impose no currency order or currency constraints on I_t. Note that each completion I_t^c yields a current tuple $\mathsf{LST}(I_t^c)$ that encodes a truth assignment μ_X of X in its first n attributes.

The set Γ of constant CFDs is given as follows. For each clause C_j, we define $\psi_j = t_p[L_1, L_2, L_3] \rightarrow t_p[C]$ where $L_i = X_k$ if ℓ_i^j or $\bar{\ell}_i^j$ is x_k and the pattern tuple $t_p = (v_1, v_2, v_3, 1)$ is given by $v_i = 1$ if $\ell_i^j = x_k$ and $v_i = 0$ if $\ell_i^j = \bar{x}_k$. Clearly, a completion $I_t^c \models \psi_j$ if the truth assignment μ_X encoded by the current tuple $\mathsf{LST}(I_t^c)$ makes C_j true. We further add $\psi_C = s_p[C] \rightarrow s_p[C]$ with $s_p = (1,0)$ to Γ, which intuitively prevents any clause to be satisfied. Indeed, a completion I_t^c such that $I_t^c \models \psi_C$ must set the C-attribute of its current tuple to 0. Contrast this with the requirement on the C-attribute of current tuples imposed by the ψ_j's.

We next show the correctness of the reduction. If φ is a tautology then every truth assignment μ_X makes at least one clause C_j true. That is, any valid I_t^c must set the C-attribute of its current to 1 (by ψ_j) and at the same time it must set the C-attribute to 0 (by ψ_C). Hence, no valid completion can exists. Conversely, if there exists a valid completion I_t^c of S_e such that $I_t^c \models \Gamma$, then its current tuple must have its C-attribute set to 0. In other words, none of the left-hand sides of the ψ_j's can be true, and hence μ_X must make all clauses false. In other words, μ_X is a counterexample to the validity of φ and hence φ is not a tautology. □

Implication. The second problem aims to deduce partial temporal orders that are logical consequences of the given currency order and currency constraints. Consider a valid specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity e and a partial temporal order $O_t = (I_e, \preccurlyeq'_{A_1}, \ldots, \preccurlyeq'_{A_n})$. We say that O_t is *implied by* S_e, denoted by $S_e \models O_t$, iff *for all* valid completions I_t^c of S_e, $O_t \subseteq I_t^c$. Here $O_t \subseteq I_t^c$ if $\preccurlyeq'_{A_i} \subseteq \preccurlyeq^c_{A_i}$ for all $i \in [1,n]$, where $I_t^c = (I_e, \preccurlyeq^c_{A_1}, \ldots, \preccurlyeq^c_{A_n})$.

The *implication problem* for conflict resolution is to decide, given a valid specification S_e and a partial temporal order O_t, whether $S_e \models O_t$.

That is, no matter how we complete the temporal instance I_t of S_e, as long as the completion is valid, the completion includes O_t in its currency orders. The implication analysis is conducted at step (2) of the framework of Fig. 5.4, for deducing true values of attributes.

Unfortunately, this problem is also intractable.

Theorem 5.4.2: *The implication problem for conflict resolution is coNP-complete. It remains coNP-hard for valid specifications* $S_e = (I_t, \Sigma, \Gamma)$ *of an entity when (1) both* Σ *and* Γ *are fixed; (2)* $\Gamma = \emptyset$*; or (3)* $\Sigma = \emptyset$*.* □

Proof. [†] The coNP upper bound is verified by providing an NP algorithm for the complement problem. In a nutshell, given a specification $S_e = (I_t, \Sigma, \Gamma)$ and a partial temporal order O_t, the algorithm simply guesses a completion I_t^c of I_t and then verifies whether (i) $I_t^c \models \Sigma$; (ii) $I_t^c \models \Gamma$; and (iii) $O_t \not\subseteq I_t^c$. If I_t^c passes these checks successfully, then the algorithm returns "yes" since $S_e \not\models O_t$. Otherwise, the current guess is rejected. This is clearly an NP algorithm for the complement problem and hence the implication problem is in coNP.

For the lower bounds, we show that the implication problem is coNP-hard when (1) both Σ and Γ are fixed; (2) $\Gamma = \emptyset$; or (3) $\Sigma = \emptyset$. The lower bounds for (1) and (2) are established by a revision of the proof of Theorem 5.4.1. More specifically, we revise the reduction used there as follows. First, the relation schema used in that proof is extended with an additional attribute A. Second, each tuple t in the temporal instance I_t has now two copies: a tuple t^a with its A-attribute set to a constant a, *i.e.,* $t^a[A] = a$, and a tuple t^b with its $t^b[A] = b$. Finally, the premise of each currency constraint used in that proof carries an additional condition "$t_1[A] = a \wedge t_2[A] = b \wedge t_1[A] \preccurlyeq_A t_2[A]$". These conditions enforce the constraints to have an effect only on completions in which b is more current than a in attribute A.

[†] This proof is a joint work with Floris Geerts.

Denote by $S'_e = (I'_t, \Sigma', \Gamma = \emptyset)$ the specification obtained from S_e in the proof of Theorem 5.4.1 after such revisions. Let O_t be the partial temporal order $(I'_t, \{t^b \preccurlyeq_A t^a\}, \emptyset, \ldots, \emptyset)$, where t^a and t^b are the two copies of an arbitrary tuple t in I_t. We claim the following: (i) S'_e is valid; and (ii) $S'_e \models O_t$ iff the formula φ is not satisfiable. For (i) it suffices to observe that for any completion $(I'_t)^c$, as long as it puts $t^b \preccurlyeq_A t^a$ in its currency order for A and arbitrarily completes currency orders for all the other attributes, it makes a valid completion. Indeed, this is simply because the conditions added to the premise of constraints used in the proof of Theorem 5.4.1 are false, and hence the currency constraint vacuously hold. Hence, S'_e is valid.

For (ii), assume first that there exists a truth assignment μ_X that makes φ true. We define a completion $(I'_t)^c$ of I'_t by setting $t^a \preccurlyeq_A t^b$, where t is the tuple used to define O_t, and by completing the currency orders for the attributes based on μ_X as in the proof of Theorem 5.4.1. As a result, $O_t \not\subseteq (I'_t)^c$ and $S'_e \not\models O_t$. Conversely, suppose that $S'_e \not\models O_t$. This implies the existence of a valid completion $(I'_t)^c$ of I'_t that includes $t^a \preccurlyeq_A t^b$ and satisfies all currency constraints in Σ'. Similar to the proof of Theorem 5.4.1 it is readily verified that a truth assignment μ_X can be constructed from $(I'_t)^c$ that makes φ true. Hence, $S'_e \models O_t$ iff φ is not satisfiable. Observe that the proof only uses a fixed set of currency constraints and does not require any constant CFDs.

Similarly, the coNP-lower bound for (3) is established by a similar modification of the specification for its counterpart given in the proof of Theorem 5.4.1, by reduction from the tautology problem. More specifically, given an instance φ of the tautology problem as stated in the proof of Theorem 5.4.1, we extend the schema R' given there with an additional attribute A. Its temporal instance I'_t now consists of two tuples $t_0 = (a, 0, \ldots, 0)$ and $t_1 = (b, 1, \ldots, 1)$. We further extend the constant CFDs $\psi_j = t_p[L_1, L_2, L_3] \rightarrow t_p[C]$ in the proof of Theorem 5.4.1 to $\psi'_j = t'_p[A, L_1, L_2, L_3] \rightarrow t'_p[C]$, where $t'_p = (a, t_p)$. Similarly for ψ_C. That is, these constant CFDs only have an effect when the current tuple has a as its A-attribute value. Denote by $S'_e = (I'_t, \Sigma = \emptyset, \Gamma')$ the specification obtained in this way. Clearly, S'_e is consistent since we just need to enforce $t_0 \preccurlyeq^c_A t_1$ in a completion to assure that the corresponding current tuple vacuously satisfies the CFDs in Γ'. Consider $O_t = (I'_t, \{t_0 \preccurlyeq^c_A t_1\}, \emptyset, \ldots, \emptyset)$. Then, similar to the argument given above, one can readily verify that $S'_e \models O_t$ iff φ is a tautology. □

True value deduction. The third problem is the *true value problem* for conflict resolution. It is to decide, given a valid specification S_e for an entity e, whether $\mathsf{T}(S_e)$ exists. That is, there exists a tuple t_c such that for all valid completions I^c_t of S_e, $\mathsf{LST}(I^c_t) = t_c$.

This analysis is needed by step (3) of the framework (Fig. 5.4) to decide whether S_e has enough information to deduce $\top(S_e)$, *i.e.,* whether additional temporal information is needed to determine the true value of e.

No matter how important this problem is, it is also nontrivial: it is coNP-complete, and remains intractable in several practical special cases.

Theorem 5.4.3: *The true value problem for conflict resolution is coNP-complete. It remains coNP-hard for valid specifications $S_e = (I_t, \Sigma, \Gamma)$ for an entity even when (1) both Σ and Γ are fixed; (2) $\Gamma = \emptyset$; or (3) $\Sigma = \emptyset$.* □

Proof. [‡] The upper bound is verified by providing an NP algorithm for the complement problem. Given a specification $S_e = (I_t, \Sigma, \Gamma)$, the algorithm simply guesses *two* completions I_t^c and $(I_t^c)'$ of I_t and then checks whether both completions are valid and generate *different* current tuples. If so, the algorithm returns "yes" and concludes that no true value of S_e can be determined. Otherwise, the current guesses are rejected. This is clearly an NP algorithm for the complement problem, and hence the true value problem is in coNP.

For the lower bounds, we need to show that the true value problem is coNP-hard when (1) both Σ and Γ are fixed; (2) $\Gamma = \emptyset$; or (3) $\Sigma = \emptyset$. The lower bounds for (1) and (2) are verified by a modification of the proof of its counterpart for Theorem 5.4.2. Indeed, it suffices to add two tuples $t_{\#}^a = (a, \#, \ldots, \#)$ and $t_{\#}^b = (b, \#, \ldots, \#)$ to the temporal instance given there, together with additional currency constraints that enforce # to come after any other constant in the currency orders for all attributes of the schema except for A (which does not carry #). Denote by $S_e'' = (I_t'', \Sigma'', \Gamma = \emptyset)$ the specification obtained in this way from S_e' given in the proof of Theorem 5.4.2. As a consequence, any completion of S_e'' can only yields current tuples $t_{\#}^a$ or $t_{\#}^b$.

As argued there, S_e'' is valid since one only has to consider a completion that includes $t_{\#}^b \preceq_A t_{\#}^a$. Furthermore, we next show that a true value exists iff φ is not satisfiable. Indeed, suppose that φ is not satisfiable. Then for any valid completion $(I_t'')^c$ of I_t'', if $(I_t'')^c \models \Sigma''$, then it has to set $t_{\#}^b \preceq_A t_{\#}^a$. Indeed, otherwise the currency constraints will be triggered and the completion would generate a satisfying truth assignment for φ, which by assumption does not exist. Hence, the true value will be the tuple $t_{\#}^a$. Conversely, suppose that no true value exists. This implies that there exist two completions of I_t'', such that one leads to current tuple $t_{\#}^a$, and the other one leads to current tuple $t_{\#}^b$. In the second case, $t_{\#}^a \preceq_A t_{\#}^b$ and hence, as argued in the proof of Theorem 5.4.2, one

[‡] This proof is a joint work with Floris Geerts.

can construct a satisfying truth assignment for φ from the completions. Hence, if no true value exists, then φ must be satisfiable.

The coNP-lower bound for (3) is established by a modification of the specification given in the proof for the case of constant CFDs in Theorem 5.4.1, by reduction from the tautology problem. The modification is as follows. Given an instance φ of the tautology problem as stated in the proof of Theorem 5.4.1, we introduce a third tuple $t_b = (b, b, \ldots, b)$ to the temporal instance given there, and extend the set Γ' of constant CFDs by including $\psi^i_{a\neg b} = t_p[AX_i] \to t_p[A]$ with $t_p = (a, b, b)$, for $i \in [1, n]$. These constant CFDs prevent the current tuple t in completions to have $t[A] = a$ and $t[X_i] = b$ for all $i \in [1, n]$. In addition, we add $\psi^i_{bb} = s_p[A] \to s_p[X_i]$ with $s_p = (b, b)$, and $\psi_{bb} = s_p[A] \to s_p[C]$ with $s_p = (b, b)$. These assure that for all current tuples t, it $t[B] = b$, the t has the constant b in all of its attributes.

Denote by $S''_e = (I''_t, \Sigma = \emptyset, \Gamma'')$ the specification obtained this way from S'_e given in the proof of Theorem 5.4.1. A completion that results in current tuple t_b is clearly a valid completion, and hence S''_e is valid itself. Moreover, it is readily verified that a true value exists iff φ is a tautology. Indeed, observe first that completions either result in the current tuple t_b or a tuple of the form $(a, \mu_X, 0)$, where μ_X is a truth assignment for X. While t_b can always be witnessed by a valid completion of S''_e (as mentioned above), $(a, \mu_X, 0)$ can only be witnessed provided that μ_X makes φ false (using the argument given in the proof of Theorem 5.4.1). Hence t_b is the true value iff φ is a tautology. □

Coverage analysis. Finally, the *minimum coverage problem* is to determine, given a valid specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity and a positive integer k, whether there exists a partial temporal order O_t such that (1) $\mathsf{T}(S_e \oplus O_t)$ exists, and (2) $|O_t| \leq k$.

Intuitively, this is to check whether one can add a partial temporal order O_t of a *bounded* size to a specification such that the enriched specification has sufficient information to deduce all the true values of an entity. The ability to solve this problem helps us identify what *minimum* additional temporal information is needed to deduce the true value. The analysis of *minimum* O_t is required by step (4) of the framework of Fig. 5.4.

This problem is Σ^p_2-complete ($\mathsf{NP}^{\mathsf{NP}}$ or $\mathsf{NP}^{\mathsf{coNP}}$), unfortunately. Worse still, it remains Σ^p_2-hard even in several practical special cases, as stated below.

Theorem 5.4.4: *The minimum coverage problem is Σ^p_2-complete. It remains Σ^p_2-hard for valid specifications $S_e = (I_t, \Sigma, \Gamma)$ for an entity even when (1) both Σ and Γ are fixed; (2) $\Gamma = \emptyset$; or (3) $\Sigma = \emptyset$.* □

Proof. [§] For the Σ_2^p upper bound it suffices to observe that the following $\mathrm{NP}^{\mathrm{coNP}}$ algorithm correctly decides whether there exists a partial temporal order O_t of size $|O_t| \leq k$ such that $\mathsf{T}(S_e \oplus O_t)$ exists. Given a valid specification $S_e = (I_t, \Sigma, \Gamma)$, the algorithm first guesses a partial temporal order O_t and then checks whether $|O_t| \leq k$ and whether $\mathsf{T}(S_e \oplus O_t)$ exists. The latter can be done in coNP (see Theorem 5.4.3). If the guessed partial temporal order passes these checks, then the algorithm returns "yes". Otherwise, the guessed order is rejected. The algorithm is in Σ_2^p since it is a non-deterministic PTIME algorithm by calling a coNP oracle (see, *e.g.,* [Pap94] for detailed discussion about Σ_2^p).

We now show that the problem is Σ_2^p-hard when (1) Σ and Γ are fixed; (2) $\Gamma = \emptyset$; or (3) $\Sigma = \emptyset$.

For (1) and (2) we establish the Σ_2^p-lower bound by reduction from the $\exists^*\forall^*\mathsf{DNF}$ problem, which is known to be Σ_2^p-complete [Sto76]. An instance of the $\exists^*\forall^*\mathsf{DNF}$ problem is a formula of the form $\varphi = \exists X \forall Y \psi$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, $\psi = C_1 \vee \cdots \vee C_r$; for $j \in [1, r]$, $C_j = \ell_1^j \wedge \ell_2^j \wedge \ell_3^j$, and for $k = 1, 2, 3$, the literal ℓ_k^j is either a variable or the complement of a variable in $X \cup Y$. It is to determine whether φ is true.

Given an instance φ of the $\exists^*\forall^*\mathsf{DNF}$ problem, we define a specification $S_e = (I_t, \Sigma, \Gamma)$ and a constant k such that the minimal coverage problem for S_e and k has a solution iff φ is true. In particular, in S_e we have a fixed set of currency constraints and no constant CFDs. Hence, the reduction shows (1) and (2).

Recall the relation schema $R(A, D, C, P, U, V, W)$ used in the proof of Theorem 5.4.2. We populate its temporal instance $I_t = (I, \preccurlyeq_A, \preccurlyeq_D, \preccurlyeq_C, \preccurlyeq_P, \preccurlyeq_U, \preccurlyeq_V, \preccurlyeq_W)$ as follows: We assume the presence of $2(n+m)$ distinct constants a_i and b_i for $i \in [1, n]$ and c_i and d_i for $i \in [1, m]$. As in the proof of Theorem 5.4.2, truth values for variables in X are encoded by means of two tuples:

$$(a, 0, x_i, 0, a_i, a_i, a_i) \quad \text{and} \quad (a, 0, x_i, 0, b_i, b_i, b_i).$$

with their A-attribute set to a, and two tuples

$$(b, 0, x_i, 0, a_i, a_i, a_i) \quad \text{and} \quad (b, 0, x_i, 0, b_i, b_i, b_i).$$

with their A-attribute set to b. Similarly, truth values for variables in Y are encoded by the following tuples:

[§] This proof is a joint work with Floris Geerts.

$$(a,0,y_i,0,c_i,c_i,c_i) \quad \text{and} \quad (a,0,y_i,0,c_i,c_i,c_i) \quad \text{and}$$
$$(b,0,y_i,0,d_i,d_i,d_i) \quad \text{and} \quad (b,0,y_i,0,d_i,d_i,d_i).$$

Moreover, we add a currency constraint to Σ for every pair of attributes (L,L') taken from $\{U,V,W\}$:

$$\begin{aligned}\forall t_1,t_2 \in R(t_1[D]=0 \wedge t_2[D]=0 \wedge t_1[C]=t_2[C] \wedge \\ t_1[A]=a \wedge t_2[A]=b \wedge t_1[A] \prec_A t_2[A] \wedge \\ t_1[L] \prec t_2[L] \rightarrow t_1[L'] \prec t_2[L'].\end{aligned}$$

These constraints ensure that whenever $a \prec_A b$, the order between a_i and b_i (resp. c_i and d_i) is consistent for all attributes U, V and W. As before, $a_i \prec_U b_i$ indicates that x_i is set to true, whereas $b_i \prec_U a_i$ indicates that x_i is false; similarly for variables in Y but using the constants c_i and d_i instead. In other words, with every completion of I_t in which $a \prec_A b$, we can associate truth assignments μ_X and μ_Y of X and Y, respectively.

We next encode the clauses in φ in a similar way the one given in the proof of Theorem 5.4.2. More specifically, given $C_1 \vee \cdots \vee C_r$ we encode its negation $\bar{C}_1 \wedge \cdots \wedge \bar{C}_r$ with $\bar{C}_j = \bar{\ell}_1^j \vee \bar{\ell}_2^j \vee \bar{\ell}_3^j$. Such clauses can be equivalently written as $\ell_1^j \wedge \ell_2^j \rightarrow \bar{\ell}_3^j$ by means of the tuples

$$(a,1,j,1,v_1,v_2,v_3) \quad \text{and} \quad (a,1,j,2,v_1',v_2',v_3'),$$

and their b-variants

$$(b,1,j,1,v_1,v_2,v_3) \quad \text{and} \quad (a,1,j,2,v_1',v_2',v_3').$$

Here $v_i = a_k$ and $v_i' = b_k$ if $\ell_i^j = x_k$, and $v_i = b_k$ and $v_i' = a_k$ if $\bar{\ell}_i^j = x_k$, for $i = 1,2$; we define v_i and v_i' the other way around for $i = 3$; similarly for variables in Y but then using constants c_i and d_i instead. For example, consider the clause $C = x_1 \wedge \bar{x}_2 \wedge \bar{y}_3$ whose complement is $\bar{C} = \bar{x}_1 \vee x_2 \vee y_3$. Equivalently, we write $\bar{C}$ as $x_1 \wedge \bar{x}_2 \rightarrow y_3$. Hence, we encode $\bar{C}$ by $(a,1,\_,1,a_1,b_2,c_3)$ and $(a,1,\_,2,b_1,a_2,d_3)$, together with their b-counterparts $(b,1,\_,1,a_1,b_2,c_3)$ and $(b,1,\_,2,b_1,a_2,d_3)$.

The link between truth assignments selected by completions and the validity of (complemented) clauses is established by the following currency constraint:

$$\begin{aligned}\forall t_1, t_2 \in R(t_1[D] = 1 \wedge t_2[D] = 1 \wedge t_1[C] = t_2[C] \wedge \\ t_1[P] = 1 \wedge t_1[P] = 2 \wedge \\ t_1[A] = a \wedge t_2[A] = b \wedge t_1[A] \prec_A t_2[A] \wedge \\ t_1[U] \prec t_2[U] \wedge t_1[V] \prec t_2[V] \rightarrow t_1[W] \prec t_2[W].\end{aligned}$$

This constraint tells that whenever the truth assignment (represented by a completion) makes $\ell_1^j \wedge \ell_2^j$ true, then it must also make $\bar{\ell}_3^j$ true, provided that $a \prec_A b$.

We also include two tuples $t_\#^a = (a, \#, \ldots, \#)$ and $t_\#^b = (b, \#, \ldots, \#)$ in I_t, which serves as potential true values of the entity represented by S_e. We enforce the symbol # to come after any other constant in currency orders by means of currency constraints (one for each attribute in R), as defined in the proof of Theorem 5.4.3. Clearly, in valid completions, if $a \prec_A b$ then $t_\#^b$ is the current tuple; when $b \prec_A a$, $t_\#^a$ is the current tuple.

Finally, we ensure that the partial temporal order O_t can only add currency information related to the values a_i and b_i in the instance, so that O_t can only affect the choice of truth values for variables in X. To achieve this, observe that given instance I_t constructed so far, $|O_t|$ is no larger than $7|I|^2$, where 7 is simply the number of attributes in R. We let $k = 7|I|^2$. Next, for each constant v different from the a_i's and b_i's we add $p > k$ tuples of the form $(v_{\text{id}}, v, \ldots, v)$, where v_{id} is a unique identifier for each of these tuples. Let I_t' denote the temporal instance obtained in this way and let $S_e = (I', \Sigma, \Gamma = \emptyset)$. Clearly, for any O_t that relates tuples in an attribute with values different from a_i and b_i, O_t will cause the addition of more than k tuples. Indeed, let B be an arbitrary attribute. Then the addition of $t \prec_B t'$ implies that $s \prec_B s'$ for all tuples s and s' that share the same B-attribute value with t and t', respectively. By the choice of k and the addition of $p > k$ tuples for each constant, any O_t of size $\leq k$ can only relate tuples that contain a_i or b_i values in one of its attributes.

Observe that the specification S_e defined above is valid. Indeed, any completion that makes a more current than b in the A-attribute vacuously satisfies the currency constraints in Σ. As a consequence $t_\#^a$ will always be one of the possible current tuples.

We next show that the minimum coverage problem has a solution iff φ is true. Suppose first that φ is false. In other words, for every μ_X of X, there exists a truth assignment μ_Y of Y that makes $C_1 \vee \cdots \vee C_r$ false. Consider a partial temporal order O_t with $|O_t| \leq k$. By the construction, O_t can only add temporal information between tuples that concern variables in X. In other words, the impact of O_t is that it restricts the set of truth assignments of X that can be obtained by means of valid completions.

However, since φ is false, even for each μ_X in this restricted set, there exists a μ_Y that makes the $C_1 \vee \cdots \vee C_r$ false. This in turn implies that $t_{\#}^b$ can be a current tuple in a completion that sets $a \prec_A b$. Indeed, simply consider the completion that (i) sets $a \prec_A b$; (ii) selects a μ_X that belongs to the restricted set; (iii) selects μ_Y such that the clauses are false; and (iv) arbitrarily complete partial currency orders for the other attributes. It is easily verified that this completion indeed satisfies all currency constraints since it satisfies the constraints related to truth assignments and all constraints corresponding to the negated clauses (recall that μ_X and μ_Y make all $\bar{C}_j$ true). Hence, when φ is false, both $t_{\#}^a$ and $t_{\#}^b$ are current tuples and no true value can exist, no matter what O_t is.

Conversely, suppose that φ is true. That is, there exists a truth assignment μ_X of X such that for all μ_Y of Y, $C_1 \vee \cdots \vee C_r$ is true. We let O_t be the partial temporal order that restricts the choices of truth assignments for X to be μ_X. By the construction, this can be done by using $\leq k$ added pairs. Then it is impossible that $t_{\#}^b$ becomes a current tuple. Indeed, for this to happen we need a completion that sets $a \prec_A b$ and in addition satisfies all constraints in Σ. This, however, would imply the existence of a truth assignment μ_Y of Y, which, together with μ_X, makes $C_1 \vee \cdots \vee C_r$ false. This in contrast to the assumption that φ holds for μ_X. As a consequence, $\mathsf{T}(S_e \oplus O_t)$ exists and is equal to $t_{\#}^a$.

Finally, we show that the problem is Σ_2^p-hard for case (3), when $\Sigma = \emptyset$. This is verified again by reduction from the $\exists^*\forall^*\mathsf{DNF}$ problem, but now we use constant CFDs only. The idea behind the reduction is similar to that of the reduction given for cases (1) and (2).

Given an instance φ of the $\exists^*\forall^*\mathsf{DNF}$ problem, we define a relation schema $R(A, X_1, \ldots, X_n, Y_1, \ldots, Y_m, C)$. To populate its corresponding temporal instance I_t, we start with two tuples $t_0 = (a, 0, 0, \ldots, 0)$ and $t_1 = (b, 1, 1, \ldots, 1)$. Completions thus lead to current tuples ranging over all possible truth assignments for X and Y. We further introduce a tuple $t_b = (b, b, \ldots, b)$, which will correspond to the true value of the entity if it exists. Finally, let $k = n$ and add $p > n$ tuples of the form $(c_i, \ldots, c_i, 0, 0, 0, \ldots)$ and $(c_i, \ldots, c_i, 1, \ldots, 1)$ to I_t, for $i \in [1, p]$. Here the c_i's are values of the attributes A and $X_1, \ldots, X_n$. We further assume that initial temporal orders are available, asserting that the c_i's come before a, b, 0 and 1. Intuitively, the addition of these p tuples will cause any additional temporal information in the Y-attributes (and A-attribute) to have more than "k effects", *i.e.,* if $t_0 \prec_Y t_1$ is in O_t, then this addition needs to be imposed on all p tuples as well since these tuples contain the same values in their Y-attributes as t_0 and t_1. As a consequence, any partial temporal order O_t of size $\leq k$ can only enrich currency orders for the X-attributes. In other words, adding O_t will cause the selection of a truth assignment for X.

We use the same constant CFDs as those defined in the proof of Theorem 5.4.3, and let $S_e = (I_t, \Sigma = \emptyset, \Gamma)$ be the resulting specification. As argued in the proof of Theorem 5.4.3, S_e is valid because any completion of S_e with $a \prec_A b$ satisfies the CFDs in Γ. Recall also that t_b will be the current tuple in this case.

We next show that the minimum coverage problem has a solution for S_e and k iff φ is true. Indeed, suppose that φ is false. Then for all truth assignments μ_X of X, there exists a μ_Y of Y, such that $C_1 \vee \cdots \vee C_r$ is false. Let O_t be any partial temporal order of size $\leq k$. As argued above, the addition of O_t causes the selection of a subset of truth assignments of X. For any such μ_X we have a μ_Y that makes the clauses false. In other words, a completion exists, which puts (i) $b \prec_A a$; (ii) selects a μ_X; and (iii) picks μ_Y that falsifies φ. By the definition of the CFDs, this implies that a current tuple of the form $(a, \ldots)$ exists and hence there is no true value for the entity (since t_b is also a current tuple).

Conversely, if φ is true, we simple take O_t that selects the satisfying truth assignment μ_X of X such that for all μ_Y of Y, all the clauses in φ are satisfied. Such O_t can be taken of size $\leq k$. In other words, completions with $b \prec_A a$ cannot exist by the definition of the constant CFDs. Hence, t_b is the only possible current tuple and thus the true value of S_e exists. □

Remark. From these results we find the following.

(i) The main conclusion is that while these problems are important in practice, they are hard. In fact as we have shown that the lower bounds of all these problems remain intact for specifications $S_e = (I_t, \Sigma, \Gamma)$ of an entity when (1) both Σ and Γ are fixed; (2) $\Gamma = \emptyset$, *i.e.,* when constant CFDs are absent; or (3) $\Sigma = \emptyset$, *i.e.,* when currency constraints are absent. Hence unless P = NP, efficient (PTIME) algorithms for solving these problems are necessarily *heuristic*.

(ii) The results not only reveal the complexity of reasoning about conflict resolution, but also advance our understanding of data currency and consistency. Indeed, while the minimum coverage problem is specific for conflict resolution and has *not* been studied before, the other three problems are also of interest to the study of data currency. Taken together with the complexity results of [FGW11], Theorems 5.4.1, 5.4.2 and 5.4.3 show that currency constraints make our lives easier as opposed to denial constraints: they reduce the complexity of inferring data currency [FGW11], from Σ_2^p-complete, Π_2^p-complete (coNP$^{\mathsf{NP}}$) and Π_2^p-complete down to NP-complete, coNP-complete and coNP-complete, respectively,

When it comes to data consistency, it is known that the satisfiability and implication problems for general CFDs are NP-complete and coNP-complete, respectively [FGJK08]. Theorems 5.4.1 and 5.4.2 give a *stronger* result: these lower bounds already hold for constant CFDs.

5.5 Algorithms for Conflict Resolution

We next provide algorithms underlying the framework depicted in Fig. 5.4. We first present an algorithm for checking whether a specification is valid (step (1) of the framework; Section 5.5.1). We then study how to deduce true attribute values from a valid specification (step (2); Section 5.5.2). Since not all true attribute values can be deduced automatically, we further discuss algorithms to generate suggestions such that the users may solicit true values of some attributes (step (4); Section 5.5.3), which can in turn help the deduction procedure.

5.5.1 Validity Checking

We start with algorithm IsValid that, given a specification $S_e = (I_t, \Sigma, \Gamma)$, returns true if S_e is valid, and false otherwise. As depicted in Fig. 5.4, IsValid is invoked for an initial specification S_e and its extensions $S_e \oplus O_t$ with the input O_t from the users.

Theorem 5.4.1 tells us that it is NP-complete to determine whether S_e is valid. In other words, IsValid is necessarily heuristic if it is to be efficient. Instead of designing an efficient algorithm from scratch, we approach this by reducing the problem to SAT, one of the most studied NP-complete problem, which is to decide whether a Boolean formula is satisfiable (see, *e.g.,* [BHvMW09]). Several high-performance tools for SAT (SAT-solvers) are already in place [BHvMW09], which have proved effective in *e.g.,* software verification, AI and operations research. For instance, MiniSAT [GT04] can effectively solve a formula with $4,500$ variables and $100K$ clauses in 1 second.

Algorithm IsValid leverages existing SAT-solvers. We convert a given specification S_e to a propositional formula in the conjunctive normal form (CNF), and then employ an SAT-solvers to decide the satisfiability of S_e.

Algorithm. More specifically, given a specification S_e of an entity e, IsValid works in three steps as follows.

(i) Instantiation: It first expresses S_e as a set of (propositional) predicate formulas.

(ii) ConvertToCNF: It then converts the predicate formulas from (i) into a CNF such that the given specification is valid iff the CNF is satisfiable.

(iii) Finally, it applies an SAT-solver to the CNF resulted from (ii), and concludes that the input specification S_e is valid iff the CNF is satisfiable.

We next present procedures Instantiation and ConvertToCNF. Consider $S_e = (I_t, \Sigma, \Gamma)$, where $I_t = (I_e, \preccurlyeq_{A_1}, \ldots, \preccurlyeq_{A_n})$ is a temporal instance of schema R. We denote also by R the set $\{A_i \mid i \in [1,n]\}$ of attributes in R. We define the *extended active domain* of A_i, denoted by $\mathsf{adom}^v(I_e.A_i)$, to be the set including all the values in $\mathsf{adom}(I_e.A_i)$ and all the constants that appear in attribute A_i of some constant CFDs in Γ. To check whether S_e is satisfiable, it suffices to consider the values from the extended active domains only.

Instantiation. To uniformly treat partial currency orders, currency constraints and constant CFDs as predicate formulas, we introduce a notion of *instance constraints*. The set of instance constraints of S_e, denoted as $\Omega(S_e)$, is defined in terms of values in the extended active domains, and a strict partial order $\prec^v_{A_i}$ on $\mathsf{adom}^v(I_e.A_i)$. These constraints are derived from S_e as follows.

(1) *Currency orders*. To encode the partial currency orders in I_t, for each $A_i \in R$, we include the following instance constraints in $\Omega(S_e)$.

(a) Partial orders in I_t: $(true \to t_1[A_i] \prec^v_{A_i} t_2[A_i])$ for each $t_1 \preccurlyeq_{A_i} t_2$ in I_t, as long as $t_1[A_i] \neq t_2[A_i]$.

(b) Transitivity of $\prec_{A_i}$: $(a_1 \prec^v_A a_2 \wedge a_2 \prec^v_A a_3 \to a_1 \prec^v_A a_3)$ for all distinct values a_1, a_2, a_3 in $\mathsf{adom}^v(I_e.A_i)$.

(c) Asymmetry: $(a \prec^v_{A_i} b) \to \neg(b \prec^v_{A_i} a)$ for all distinct values $a, b \in \mathsf{adom}^v(I_e.A_i)$.

Intuitively, these assure that each $\prec^v_{A_i}$ is a *strict partial order* (via both (b) and (c)), and express available temporal information in I_t as predicate formulas (via (a)).

(2) *Currency constraints*. For each currency constraint $\varphi = \forall t_1, t_2\ (\omega \to t_1 \prec_{A_r} t_2)$ in Σ and for all distinct tuples $s_1, s_2 \in I_e$, we include the following in $\Omega(S_e)$:

$$\mathsf{ins}(\omega, s_1, s_2) \ \to\ s_1[A_r] \prec^v_{A_r} s_2[A_r],$$

where $\mathsf{ins}(\omega, s_1, s_2)$ is obtained from ω by (a) substituting $s_i[A_j]$ for t_i and $\prec^v_{A_j}$ for $\prec_{A_j}$ in each predicate $t_1 \prec_{A_j} t_2$, for $i \in [1,2]$; and (b) evaluating each conjunct of ω defined with a comparison operator to its truth value *w.r.t.* s_1 and s_2. Intuitively, $\mathsf{ins}(\omega, s_1, s_2)$ "instantiates" ω with values in s_1 and s_2.

Example 5.5.1: For currency constraint φ_1 in Fig. 5.3, and tuples r_1 and r_2 in Fig. 5.2 for Edith, its instance constraint is derived to be (*true*→ *working* $\prec^v_{\mathsf{status}}$ *retired*). Observe that the precondition of φ_1 is evaluated to be true on these two particular tuples, by instantiating variables of φ_1 with values in r_1 and r_2.

Similarly, from currency constraint φ_6 and tuples r_1 and r_2, we derive instance constraint (*working* $\prec^v_{\mathsf{status}}$ *retired* → *212* $\prec^v_{\mathsf{AC}}$ *415*), by replacing $\prec_{\mathsf{status}}$ with $\prec^v_{\mathsf{status}}$, and by replacing variables in φ_6 with the corresponding attribute values from r_1 and r_2. □

(3) *Constant CFDs*. For each CFD $t_p[X] \to t_p[B]$ in Γ and each $b \in \mathsf{adom}^v(I_e.B) \setminus \{t_p[B]\}$, $\Omega(S_e)$ includes

$$\psi = (\omega_X \ \rightarrow \ b \prec^v_B t_p[B]),$$

where ω_X is the conjunction of all the formulas of the form $a \prec^v_{A_j} t_p[A_j]$, where a ranges over all the values in $\mathsf{adom}^v(I_e.A_j) \setminus \{t_p[A_j]\}$, for all attributes $A_j \in X$.

Intuitively, constraint ψ asserts that if $t_p[X]$ is the true value of attributes X, then $t_p[B]$ is the true value of attribute B. Indeed, the CFD is defined on $\mathsf{LST}(I^c_t)$ for a completion I^c_t of I_t (see Section 5.2.2), and ψ assures that this semantics is enforced.

Example 5.5.2: Recall constant CFD ψ_1 from Fig. 5.3. For the entity instance E_1 of Edith, the CFD is encoded by two instance constraints given below, included in Ω_{E_1}:

$$\mathit{212} \prec^v_{\mathsf{AC}} \mathit{213} \wedge \mathit{415} \prec^v_{\mathsf{AC}} \mathit{213} \rightarrow \mathit{NY} \prec^v_{\mathsf{city}} \mathit{LA},$$
$$\mathit{212} \prec^v_{\mathsf{AC}} \mathit{213} \wedge \mathit{415} \prec^v_{\mathsf{AC}} \mathit{213} \rightarrow \mathit{SFC} \prec^v_{\mathsf{city}} \mathit{LA},$$

i.e., LA is her true city value if her true AC is *213*. □

<u>ConvertToCNF</u>. After we derive $\Omega(S_e)$, we convert the instance constraints of $\Omega(S_e)$ into a CNF $\Phi(S_e)$ as follows. We first substitute a Boolean variable $x^{A_i}_{a_1 a_2}$ for each predicate $a_1 \prec^v_{A_i} a_2$ in $\Omega(S_e)$. We then rewrite each formula of the form $(x_1 \wedge \cdots \wedge x_k \rightarrow x_{k+1})$ into equivalent $(\neg x_1 \vee \cdots \vee \neg x_k \vee x_{k+1})$. Finally, $\Phi(S_e)$ is defined to be the conjunction of all such formulas obtained from $\Omega(S_e)$, which is obviously in CNF.

One can readily verify the following (by contradiction), which justifies the reduction from the satisfiability of the specification S_e to the SATinstance $\Phi(S_e)$.

Lemma 5.5.1: *Specification S_e is valid iff its converted* CNF $\Phi(S_e)$ *is satisfiable.* □

Complexity: Observe the following. (a) The size $|\Omega(S_e)|$ of $\Omega(S_e)$ is bounded by $O((|\Sigma|+|\Gamma|)|I_t|^2+|I_t|^3)$, since encoding currency orders, currency constraints and constant CFDs is in time $O(|I_t|^3)$, $O(|\Sigma||I_t|^2)$ and $O(|\Gamma||I_t|^2)$, respectively. (b) It takes $O(|\Omega(S_e)|)$ time to convert $\Omega(S_e)$ into $\Phi(S_e)$. Hence the size of the CNF $\Phi(S_e)$ is bounded by $O((|\Sigma|+|\Gamma|)|I_t|^2+|I_t|^3)$. In practice, an entity instance I_t is typically *much smaller than* a database, and the sets Σ and Γ of constraints are also *small*. As will be seen in Section 5.6, SAT-solvers can efficiently process CNFs of this size.

5.5.2 Deducing True Values

We next present an algorithm that, given a valid specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity e, deduces *true values* for as many attributes of e as possible.

Intuitively, it is to find a maximum partial order O_d such that $S_e \models O_d$, *i.e.,* (a) for all valid completions I_t^c of S_e, $O_d \subseteq I_t^c$ (Section 5.4), and (b) for tuples $t_1, t_2 \in I_e$ and $A_i \in R$, if $S_e \models t_1 \prec_{A_i} t_2$ then $t_1 \prec_{A_i} t_2$ is in O_d.

5.5.2.1 Partial Order Deduction

To deduce true values, below we first present a heuristic approach, and then discuss an exact algorithm.

5.5.2.1.1 A heuristic approach. Given S_e, we want to deduce a maximum partial order O_d such that $S_e \models O_d$. As an immediate corollary of Theorem 5.4.2, one can show that the decision problem of this problem is also coNP-complete, even when either Σ or Γ is fixed or absent. Thus we give a heuristics to strike a balance between its complexity and accuracy. The algorithm is based on the following lemma, which is easy to verify.

Lemma 5.5.2: *For the* CNF *$\Phi(S_e)$ converted from a valid specification S_e, and for all tuples t_1, t_2 in S_e such that $t_1[A_i] = a_1$ and $t_2[A_i] = a_2$, $S_e \models t_1 \prec_{A_i} t_2$ iff $\Phi(S_e) \rightarrow x^{A_i}_{a_1 a_2}$ is a tautology, where $x^{A_i}_{a_1 a_2}$ is the variable denoting $a_1 \prec^v_{A_i} a_2$ in $\Phi(S_e)$.* □

Observe that the condition $\Phi(S_e) \rightarrow x^{A_i}_{a_1 a_2}$ indicates that for any truth assignment μ, if μ satisfies $\Phi(S_e)$, then $\mu(x^{A_i}_{a_1 a_2})$ is *true*. That is, the one-literal clause $x^{A_i}_{a_1 a_2}$ is implied by $\Phi(S_e)$, which in turn encodes S_e. Based on this observation, our algorithm checks one-literal clauses in $\Phi(S_e)$ one by one, and enriches the known partial order accordingly.

Algorithm DeduceOrder

Input: A valid specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity.
Output: A partial temporal order O_d such that $S_e \models O_d$.

```
1.  $\Omega(S_e) := \mathsf{Instantiation}(S_e)$;
2.  $\Phi(S_e) := \mathsf{ConvertToCNF}(\Omega(S_e))$;
3.  $O_d := (I_e, \emptyset, \ldots, \emptyset)$;
4.  while there exists a one-literal clause $C$ in $\Phi(S_e)$ do
        /* $x^A_{a_1 a_2}$ in $C$ is the variable denoting $a_1 \prec^v_A a_2$ */
5.      if $C$ is a one-literal clause $(x^A_{a_1 a_2})$ then
6.          add $a_1 \prec^v_A a_2$ to $O_d$;
7.          $C_\neg := \neg x^A_{a_1 a_2}$;
8.      if $C$ is a one-literal clause $(\neg x^A_{a_1 a_2})$ then
9.          add $a_2 \prec^v_A a_1$ to $O_d$;
10.         $C_\neg := x^A_{a_1 a_2}$;
11.     for each $C' \in \Phi(S_e)$ do
12.         if $C'$ contains $C_\neg$ then
13.             $C' := C' \setminus C_\neg$;
14.         if $C'$ contains $C$ then
15.             Remove $C'$ from $\Phi(S_e)$;
16. return $O_d$.
```

Figure 5.5: Algorithm DeduceOrder

Algorithm. The algorithm for deducing true values, referred to as DeduceOrder, is given in Fig. 5.5. It first converts a specification S_e to a CNF $\Phi(S_e)$ (lines 1-2; see Section 5.5.1). For each literal C of the form $x^{A_i}_{a_1 a_2}$ or $\neg x^{A_i}_{a_1 a_2}$, it checks whether C is a clause in (*i.e.,* implied by) $\Phi(S_e)$ (line 4); and if so, it will enrich the partial order (lines 5-10). It then reduces $\Phi(S_e)$ by using C and its negation $C_\neg$ (lines 11-15). That is, for each clause C' that contains C, the entire C' is removed since C' is *true* if C has to be satisfied (lines 12-13). Similarly, for each clause C'' that contains $C_\neg$, $C_\neg$ is removed from C'', as $C_\neg$ has to be *false* (lines 14-15). The deduced partial order is then returned (line 16).

Example 5.5.3: Given the entity instance E_2 of Fig. 5.2 and the constraints of Fig. 5.3, DeduceOrder finds O_d including: (1) *0* $\prec^v_{\mathsf{kids}}$ *2* by φ_4, (2) *working* $\prec^v_{\mathsf{status}}$ *retired* by φ_1, (3) *sailor* $\prec^v_{\mathsf{job}}$ *veteran*, *401* $\prec^v_{\mathsf{AC}}$ *212* and *02840* $\prec^v_{\mathsf{zip}}$ *12404*, by (2) and φ_5, φ_6 and φ_7, respectively. A current tuple of George is then of the form (*George*, x_{status},x_{job}, *2*, x_{city},x_{AC},x_{zip},x_{county}), with variables.

Assume that the users assure that the true value of the attribute status is *retired*. Then the algorithm can deduce the following from the extended specification:

(a) x_{job}, x_{AC} and x_{zip} as *n/a*, *212* and *12404*, from tuple r_5 via currency constraints φ_5, φ_6 and φ_7, respectively;

(b) $x_{\mathsf{city}} = NY$, from the true value of AC (*i.e., 212* deduced in step (a) above) and the constant CFD ψ_2;

(c) x_{county} as *Accord*, from φ_8 and the true values of city and zip deduced in steps (b) and (a), respectively.

The automated deduction tells us that the true value for George is t_2 = (George, retired, n/a, 2, NY, 212, 12404, Accord). This shows that inferences of *currency constraints* help *consistency* inference (from step (a) to (b)), and *vice versa* (*e.g.,* from (b) to (c)). □

Complexity. (1) It takes $O((|\Sigma|+|\Gamma|)|I_t|^2+|I_t|^3)$ time to convert S_e into $\Phi(S_e)$ (lines 1-2; see Section 5.5.1). (2) The *total time* taken by the **while** loop (lines 4-15) is in $O((|\Sigma|+|\Gamma|)|I_t|^2+|I_t|^3)$. Indeed, we maintain a hash-based index for literals C, in which the key is C and its value is the list of clauses in $\Phi(S_e)$ that contain C or $C_\neg$. In the process, $\Phi(S_e)$ decreases monotonically. Hence in total it takes at most $O(|\Phi(S_e)|)$ time to reduce $\Phi(S_e)$ for *all literals*, where $|\Phi(S_e)|$ is bounded by $O((|\Sigma|+|\Gamma|)|I_t|^2+|I_t|^3)$. Putting these together, the algorithm is in $O((|\Sigma|+|\Gamma|)|I_t|^2+|I_t|^3)$ time.

5.5.2.1.2 An exact approach. By Lemma 5.5.2, one might want to compute a temporal order O'_d consisting of all such variables $x^{A_i}_{a_1a_2}$ that $\Phi(S_e)\wedge\neg x^{A_i}_{a_1a_2}$ is *not* satisfiable. That is, for each variable $x^{A_i}_{a_1a_2}$, we inspect the satisfiability of $\Phi(S_e)\wedge\neg x^{A_i}_{a_1a_2}$ by invoking an SAT-solver.

This approach, referred to as NaiveDeduce, is given in Fig. 5.6. It first converts specification S_e to CNF $\Phi(S_e)$ (lines 1-2). For each attribute A (line 4), it then enumerates values pair a_1,a_2 in $\mathsf{adom}^v(A)$ (line 5). It examines whether $\Phi(S_e)\rightarrow x^{A}_{a_1a_2}$

```
Algorithm NaiveDeduce

Input: A valid specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity.
Output: A partial temporal order $O_d$ such that $S_e \models O_d$.

1. $\Omega(S_e) :=$ Instantiation$(S_e)$;
2. $\Phi(S_e) :=$ ConvertToCNF$(\Omega(S_e))$;
3. $O_d := (I_e, \emptyset, \ldots, \emptyset)$;
4. for each $A \in R$ do
5.     for each $a_1, a_2 \in \mathsf{adom}^v(A)$ do
         /* $x^A_{a_1 a_2}$ is the variable denoting $a_1 \prec^v_A a_2$ */
6.       if $\neg$ SAT($\Phi(S_e) \cup \{\neg x^A_{a_1 a_2}\}$) then
7.          add $a_1 \prec^v_A a_2$ to $O_d$;
8. return $O_d$.
```

Figure 5.6: Algorithm NaiveDeduce

is a tautology by invoking the SAT-solver to check whether $\Phi(S_e) \wedge \neg x^A_{a_1 a_2}$ is not satisfiable (line 6). If $\Phi(S_e) \to x^A_{a_1 a_2}$ is a tautology, it adds $a_1 \prec^v_A a_2$ to O_d (line 7). The procedure returns O_d when all the possible partial orders are examined (line 8).

NaiveDeduce is an exact algorithm for deducing O_d provided the the SAT-solver it invokes is an exact algorithm. However, NaiveDeduce calls the SAT-solver $|I_t|^2$ times. As will be seen in Section 5.6, DeduceOrder finds O_d with its accuracy comparable to O'_d, without incurring the cost of repeatedly calling an SAT-solver.

5.5.2.2 True Value Deduction

Using the partial temporal order O_d found by DeduceOrder or NaiveDeduce, one can readily deduce true attributes values as follows: a value a_1 is the true value of attribute A_i if for all values $a_2 \in \mathsf{adom}^v(I_e.A_i) \setminus \{a_1\}$, the currency order $a_2 \prec^v_A a_1$ is in O_d.

5.5.3 Generating Suggestions

True value deduction procedures given in the previous section find us the true values $\mathsf{V}_{\mathcal{B}}$ for a set of attributes $\mathcal{B} \subseteq R$. To identify the true value of the entity e specified by $S_e = (I_t, \Sigma, \Gamma)$, instead of asking the users for input on all those attributes whose true values remain all unknown, we compute a suggestion for a set of attributes $\mathcal{A} \subseteq R$ such that if the true values for $\mathcal{A}$ are validated, the true value of *the entire e* can be determined, even for attributes in $R \setminus (\mathcal{B} \cup \mathcal{A})$ (see Fig. 5.4).

Below we first formally define suggestions and a notion of derivation rules (Section 5.5.3.1). We then provide an algorithm for computing suggestions (Section 5.5.3.2).

5.5.3.1 Suggestions and Derivation Rules

For each attribute $A_i \in R \setminus \mathcal{B}$, we denote by $\mathsf{V}(A_i)$ the set of *candidate true values for* A_i, *i.e.,* for any candidate $a_1 \in \mathsf{V}(A_i)$, there exists no $a_2 \in \mathsf{adom}^v(I_e.A_i) \setminus \{a_1\}$ such that $a_1 \prec_A^v a_2$ is in O_d. For a set X of attributes, we write $\mathsf{V}(X) = \{\mathsf{V}(A_i) \mid A_i \in X\}$.

Suggestion. A *suggestion* for S_e is a pair $(\mathcal{A}, \mathsf{V}(\mathcal{A}))$, where $\mathcal{A} = (A_1, \ldots, A_m)$ is a set of attributes of R such that $\mathcal{A} \cap \mathcal{B} = \emptyset$ and moreover, (1) there exist values $(a_1, \ldots, a_m)$ such that if $(a_1, \ldots, a_m)$ are validated as the true values of $\mathcal{A}$, then the true value $\mathsf{T}(S_e)$ of S_e exists; and (2) for all possible values $(a'_1, \ldots, a'_m)$ that satisfy condition (1), a'_i is in $\mathsf{V}(A_i)$ for $i \in [1, m]$.

Intuitively, condition (1) says that when the true values of $\mathcal{A}$ are validated, so is $\mathsf{T}(S_e)$. That is, the true values of attributes in $\mathcal{A}' = R \setminus (\mathcal{B} \cup \mathcal{A})$ can be automatically deduced from $\mathsf{V}_{\mathcal{B}}$ and the true values of $\mathcal{A}$. Condition (2) says that $\mathsf{V}(\mathcal{A})$ gives "complete" candidates for the true values of $\mathcal{A}$ from their active domains.

One naturally wants a suggestion to be as "small" as possible, so that it takes the users minimal efforts to validate the true values of $\mathcal{A}$. This motivates us to study the *minimum suggestion problem*, which is to find a suggestion $(\mathcal{A}, \mathsf{V}(\mathcal{A}))$ with the minimum number $|\mathcal{A}|$ of attributes. Unfortunately, this problem is Σ_2^p-complete ($\mathsf{NP}^{\mathsf{NP}}$), which can be verified by reduction from the minimum coverage problem (Theorem 5.4.4).

Corollary 5.5.3: *The (decision version of) minimum suggestion problem for conflict resolution is Σ_2^p-complete.* □

Proof.[¶] It suffices to observe that a solution of the minimal suggestion problem of size ℓ relates to a solution of the minimal coverage problem of size $k = \ell|I|^2$. Conversely, one can show that a solution of the minimal coverage problem of size k relates to a

[¶] This proof is a joint work with Floris Geerts and Wenfei Fan.

solution of the minimal suggestion problem of size $\lceil k/|I|^2 \rceil$. □

In light of the high complexity, we develop an effective heuristic algorithm to compute suggestions. To do this, we examine how true values are inferred by using currency constraints and constant CFDs in a specification S_e, by expressing them as a uniform set of rules.

Derivation rules. A *true-value derivation rule* for S_e has the form $(X, P[X]) \to (B, b)$, where (1) X is a set of attributes, B is a single attribute, and (2) b is a value that is either in $\mathsf{adom}(I_e.B)$ or in attribute B of some constant CFD; and (3) for each $A_i \in X$, $P[A_i]$ is drawn from $\mathsf{adom}^v(I_e.A_i)$. It assures if $P[X]$ is the true value of X, then b is the true value of B.

Derivation rules are computed from instance constraints $\Omega(S_e)$ of S_e, which is illustrated below and will be elaborated in Section 5.5.3.2.

Example 5.5.4: Sample rules for George in Fig. 5.2 include:

n_1 : ({status}, {*retired*}) → (job, *veteran*)
n_2 : ({status}, {*retired*}) → (AC, *212*)
n_3 : ({status}, {*retired*}) → (zip, *12404*)
n_4 : ({city, zip}, {*NY*, *12404*}) → (county, *Accord*)
n_5 : ({AC}, {*212*}) → (city, *NY*)
n_6 : ({status}, {*unemployed*}) → (job, *n/a*)
n_7 : ({status}, {*unemployed*}) → (AC, *312*)
n_8 : ({status}, {*unemployed*}) → (zip, *60653*)
n_9 : ({city, zip}, {*Chicago*, *60653*})
→ (county, *Bronzeville*)

Here rule n_5 is derived from CFD ψ_2, which states that if his true AC is *212*, then his true city must be *NY*. Rule n_1 is from tuple r_5 and constraint φ_5 (Fig. 5.3), which states that if his true status is *retired*, then his true job is *veteran*. Note that in n_1, status is instantiated with *retired*. Similarly, n_6 is derived from r_6 and φ_5; n_2 and n_3 (resp. n_7 and n_8) are derived from tuple r_5 (resp. r_6) and constraints φ_6 and φ_7, respectively; and n_4 (resp. n_9) is derived from r_5 (resp. r_6) and φ_8. □

To find a suggestion, we want to find a set $\mathcal{A}$ of attributes so that a maximum number of derivation rules can be applied to them at the same time. As a consequence, the true values of as many other attributes as possible can be derived from these rules. To capture this idea, we introduce the following notion.

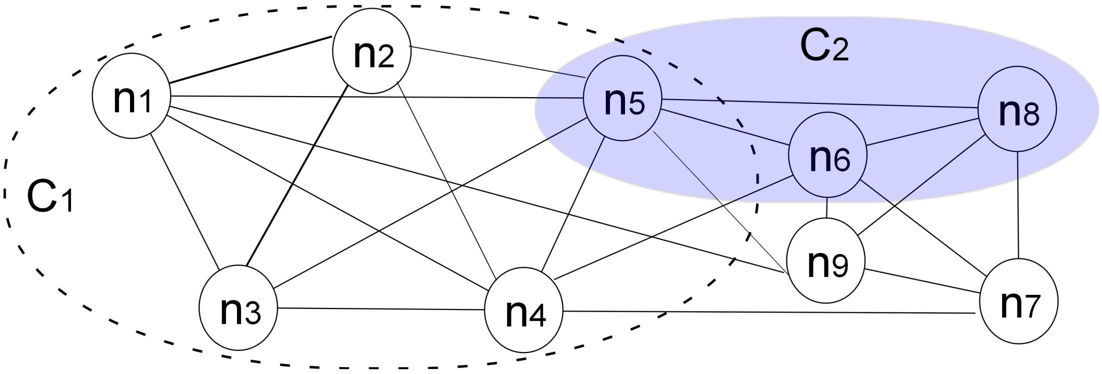


Figure 5.7: Sample compatibility graph

Compatibility graphs. Consider a set Π of derivation rules. The *compatibility graph* $G(N,E)$ of Π is an *undirected* graph, where (1) each node x in N is a rule $(X_x, P_x[X_x]) \rightarrow (B_x, b_x)$ in Π, and (2) an edge (x,y) is in E iff $B_x \neq B_y$ and $P_x[X_{xy}] = P_y[X_{xy}]$, where $X_{xy} = (X_x \cup B_x) \cap (X_y \cup B_y)$.

Intuitively, two nodes are connected (*i.e.,* compatible) in G if their associated derivation rules derive different attributes (*i.e.,* $B_x \neq B_y$), and they agree on the values of their common attributes (*i.e.,* $P_x[X_{xy}] = P_y[X_{xy}]$). Hence these rules have no conflict with each other and can be applied at the same time.

Example 5.5.5: The compatibility graph of the rules given in Example 5.5.4 is shown in Fig. 5.7. There is an edge (n_1, n_2) in the graph since their common attribute status has the same value *retired*; similarly for the other edges. In contrast, there is no edge between n_5 and n_7 since the values of their common attribute AC are different: *212* for n_5 and *312* for n_7. □

Observe that each clique $\mathcal{C}$ in the compatibility graph indicates a set of derivation rules that can be applied together. Let $\mathcal{A}'$ be the set of attributes whose true values can be derived from the rules in $\mathcal{C}$, if $\mathcal{C}$ and S_e have no conflicts (will be discussed shortly). To find a suggestion, we compute a maximum clique $\mathcal{C}$ from the graph, and derive a suggestion as $(\mathcal{A}, \mathsf{V}(\mathcal{A}))$ from $\mathcal{C}$, where $\mathcal{A}$ consists of attributes in $R \setminus (\mathcal{A}' \cup \mathcal{B})$, and $\mathsf{V}(\mathcal{A})$ is the set of candidate true values for $\mathcal{A}$.

Example 5.5.6: Example 5.2.3 shows that for George with entity instance E_2, only the true values of name and kids are known, *i.e.,* $\mathcal{B} = \{\text{name}, \text{kids}\}$ and $\mathsf{V}_{\mathcal{B}} = ($*George*, *2*$)$. To find a suggestion for George, we identify a clique $\mathcal{C}_1$ with five nodes n_1–n_5 in the compatibility graph of Fig. 5.7. Observe the following. (a) The values of job, AC and zip depend on the value of status by rules n_1, n_2 and n_3, respectively. (b)

The value of AC in turn decides city by n_5. (c) From city and zip one can derive county by n_4. Hence, the set of attributes that can be derived from clique C_1 is $\mathcal{A}' = \{\mathsf{job}, \mathsf{AC}, \mathsf{zip}, \mathsf{city}, \mathsf{county}\}$. This yields a suggestion $(\mathcal{A}, \mathsf{V}(\mathsf{status}))$, where $\mathcal{A} = R \setminus (\mathcal{A}' \cup \mathcal{B}) = \{\mathsf{status}\}$, and $\mathsf{V}(\mathsf{status}) = \{\mathit{retired}, \mathit{unemployed}\}$. As long as users identify the true value of status, the true value of George exists, and can be automatically deduced as described in Example 5.5.3. □

However, a clique C and S_e may still have conflicts and as a result, C may not yield a valid completion of S_e, as illustrated by the example below.

Example 5.5.7: Consider the clique C_2 shown in Fig. 5.7 with three nodes n_5, n_6 and n_8. Observe the following: (a) n_5 indicates that *312* $\prec^v_{\mathsf{AC}}$ *212*, since *212* is *assumed* the latest AC value; whereas (b) n_6, n_8 and constraint φ_6 in Fig. 5.3 state that *312* is the latest AC value, *i.e., 212* $\prec^v_{\mathsf{AC}}$ *312.* These tell us that the values embedded in clique C_2 may not lead to a valid completion for entity instance E_2, *i.e.,* C_2 and S_e have conflicts. □

To handle conflicts between C and S_e, we use MaxSat to find a maximum subgraph C' of C that has no conflicts with S_e (MaxSat is to find a maximum set of satisfiable clauses in a Boolean formula; see *e.g.,* [SK04]). For instance, for clique C_2 of Example 5.5.7, we use a MaxSat-solver [SK04] to identify clique C_2' with nodes n_6 and n_8, which has no conflicts with the specification for George. We then derive $\mathcal{A}' = \{\mathsf{job}, \mathsf{zip}\}$ from C_2'. Since $\mathcal{B}$ is $\{\mathsf{name}, \mathsf{kids}\}$ (Example 5.5.6), we find $\mathcal{A} = R \setminus (\mathcal{A}' \cup \mathcal{B}) = \{\mathsf{status}, \mathsf{city}, \mathsf{AC}, \mathsf{county}\}$ for a suggestion.

5.5.3.2 Computing Suggestions

We are now ready to present the algorithm for computing suggestions, referred to as $\mathsf{Suggest}$, which is shown in Fig. 5.8. It takes as input a specification S_e of e, partial orders O_d deduced from S_e ($S_e \models O_d$, by Algorithm DeduceOrder), and the set $\mathsf{V}_{\mathcal{B}}$ of validated true values. It finds and returns a suggestion $(\mathcal{A}, \mathsf{V}(\mathcal{A}))$.

Algorithm $\mathsf{Suggest}$ first computes candidate true values for all attributes whose true values are yet unknown (line 1). It then deduces a set of derivation rules from instance constraints $\Omega(S_e)$ (line 2) of S_e (line 3; as illustrated in Example 5.5.4). Based on these derivation rules, it builds a compatibility graph (line 4; see Example 5.5.5) and identifies a maximum clique C in the graph (line 5). Finally, it generates a suggestion using the clique (line 6; see Examples 5.5.6 and 5.5.7).

We next present the details of the procedures used in algorithm $\mathsf{Suggest}$ one by

```
Algorithm Suggest

Input: A specification $S_e = (I_t, \Sigma, \Gamma)$ of an entity,
       order $O_d$ ($S_e \models O_d$), and $\mathcal{V}_{\mathcal{B}}$.
Output: A suggestion $(\mathcal{A}, \mathsf{V}(\mathcal{A}))$.

1. $\mathsf{V}(R) :=$ DeriveVR$(I_t, O_d)$;
2. $\Omega(S_e) :=$ Instantiation$(S_e)$;
3. $\Pi :=$ TrueDer$(\Omega(S_e), \mathsf{V}(R))$;
4. $G :=$ CompGraph$(\Pi, S_e)$;
5. $\mathcal{C} :=$ MaxClique$(G)$;
6. $\mathcal{A} :=$ GetSug$(S_e, \mathcal{C}, \Omega(S_e), \mathcal{V}_{\mathcal{B}})$;
7. return $(\mathcal{A}, \mathsf{V}(\mathcal{A}))$;
```

Figure 5.8: Algorithm Suggest

one.

5.5.3.2.1 DeriveVR**:** For each attribute A whose true value is yet unknown, it computes a set $\mathsf{V}(A)$ of candidate true values for A. Observe that given an attribute A, for any value a_1 in its active domain $\mathsf{adom}^v(A)$, if there exists another value a_2 also from $\mathsf{adom}^v(A)$, such that $a_1 \prec_A^v a_2$, then a_1 must not be the true value for A. In other words, a_1 is known not to be the most current value.

Based on this observation, DeriveVR works as follows (not shown). Initially, $\mathsf{V}(A)$ takes the active domain $\mathsf{adom}^v(I_e.A)$. It then removes all values a_1 in $\mathsf{adom}^v(I_e.A)$ from $\mathsf{V}(A)$ if there exists a value a_2 in $\mathsf{adom}^v(I_e.A) \setminus \{a_1\}$ such that $a_1 \prec_A^v a_2$ is in the deduced partial order O_d, as a_2 is more current than a_1 in A. DeriveVR takes $O(|I_t|^2)$ time with an index, since it checks at most $|O_d|$ partial orders, and $|O_d| \leq |I_t|^2$.

5.5.3.2.2 TrueDer**:** Given a set $\Omega(S_e)$ of instance constraints, procedure TrueDer deduces a set Π of derivation rules following the same way as shown in Example 5.5.4.

(1) From constant CFDs, the derivation rules could be deduced directly as long as they do not have conflicts with the candidate true values derived by DeriveVR.

(2) From those instance constraints that represent currency constraints and currency orders, it deduces derivation rules of the form $(X, P(X)) \rightarrow (B, b)$, for each attribute B whose true value is unknown and for each $b \in \mathsf{V}(B)$, if such a rule exists. While it is

```
Procedure TrueDer

Input: A specification $S_e$ of an entity,
       a set of instance constraints $\Omega(S_e)$, and
       the candidate true values $\mathsf{V}(R)$.
Output: A set $\Pi$ of derivation rules.

$\Pi := \emptyset$ ;
for each $(t_p[X_\varphi] \rightarrow t_p[B_\varphi]) \in \Gamma$ do
  if for all attribute $A \in X_\varphi \cap \mathcal{B}, A(t_p[A] \in V(A))$ then
    $\Pi := \Pi \cup \{(X_\varphi, t_p[X_\varphi]) \rightarrow (B_\varphi, t_p[B_\varphi])\}$;
for each $B \in R$ do
  for each $b \in V(B)$ do
    $U_{(B,b)} := \{b_i \prec^v_B b \mid b_i \in \mathsf{V}(B) \setminus \{b\}\}$;
    $\Omega_{(B,b)} := \{\phi \mid \phi \in \Omega(S_e) \wedge \phi = \omega \rightarrow b_i \prec^v_B b\}$;
    $X := \emptyset; P(X) := nil;$ success $:= true$;
    for each $b_i \in U_{(B,b)}$ do
      pick a rule $\phi$ from $\Omega_{(B,b)}$ for $b_i \prec_B b$;
      if $P(X)$ satisfies $\phi$ then
        populate $X, P(X)$;
      else success :=false; break;
    if success then $\Pi := \Pi \cup \{(X, P(X)) \rightarrow (B, b)\}$;
return $\Pi$;
```

Figure 5.9: Procedure TrueDer

prohibitively expensive to enumerate all these rules, we use a heuristics to find a set of derivation rules. For each candidate true value b, it first identifies instance constraints that could complement the missing partial orders when assuming b as true value. Then it maintains and populates pattern $(X, P(X))$ from unknown attributes and candidate true values that could satisfy the premise of each of those instance constraints.

For example, the rule n_1 in Example 5.5.4 could be deduced as follows. Observe that with φ_5 (Fig. 5.3) and r_5, r_6 (Fig. 5.2), the instance constraint

$$(\text{unemployed} \prec^v_{\mathsf{status}} \text{retired} \rightarrow \text{n/a} \prec^v_{\mathsf{job}} \text{veteran})$$

could be derived. Here $\mathsf{V}(\mathsf{job})$ consists of 2 values “n/a” and “veteran”. The users may inspect the two values and choose one from the two as the true value of $\mathsf{V}(\mathsf{job})$. If one wants to assume “veteran” as the true value, “n/a” $\prec^v_{\mathsf{job}}$ “veteran” is missing from the

```
Procedure CompGraph

Input: A set $\Pi$ of derivation rules, and a specification $S_e$.
Output: A compatibility graph $G$ of $\Pi$.

initialize $G$ to be an empty graph;
for each derivation rule $n \in \Pi$ do
    add a node in $G$ for $n$;
for each node $n_i \in G$ do
    for each node $n_j \in G$ where $n_i \neq n_j$ do
        if $n_i$ and $n_j$ are compatible then
            add an edge $(n_i, n_j)$ to $G$;
return $G$;
```

Figure 5.10: Procedure CompGraph

partial order. Nonetheless, this can be complemented with the instance constraint given above. In light of this, we populate X as $\{\mathsf{status}\}$, and $P(X)$ as "retired" ($\in \mathsf{V}(\mathsf{status})$) to satisfy the premise of the constraint. When $(X, P(X))$ is in place, the derivation rule $n_1 : (\{\mathsf{status}\}, \{\mathit{retired}\}) \rightarrow (\mathsf{job}, \mathit{veteran})$ can be deduced.

More specifically, procedure TrueDer is given in Fig. 5.9. It works as follows, starting with an empty set Π of derivation rules (line 1).

(1) Deduce rules from CFDs: for each constant CFD $(t_p[X_\varphi] \rightarrow t_p[B_\varphi]) \in \Gamma$ (line 2), if $t_p[A] \in \mathsf{V}[A]$ for each $A \in X_\varphi \cap \mathcal{B}$, *i.e.,* when the values of the CFD have no conflict with those candidate true values (line 3), then we add $(X_\varphi, t_p[X_\varphi]) \rightarrow (B_\varphi, t_p[B_\varphi])$ as a new derivation rule (line 4).

(2) Deduce rules from those instance constraints in $\Omega(S_e)$ that represent currency constraints and partial currency orders in S_e, as follows:

   (i) for each attribute B whose true value is unknown (line 5) and each value b in $\mathsf{V}(B)$ that can possibly be its true value (line 6), let $U_{(B,b)} = \{b_i \prec^v_B b \mid b_i \in \mathsf{V}(B) \setminus \{b\}\}$, which is the set consisting of all the missing partial orders when b is assumed to be the true value of B (line 7);

   (ii) partition instance constraints based on $U_{(B,b)}$: for each value b in $U_{(B,b)}$, let $\Omega_{(B,b)}$ consist of all instance constraints $\phi \in \Omega(S_e)$ such that ϕ is of the form $\omega \rightarrow b_i \prec^v_B b$ (line 8); note that each ϕ appears in at most one of the partitions;

(iii) for each $b_i \in U_{(B,b)}$ (line 10), we pick a rule $\phi = \omega \rightarrow b_i \prec_B^v b$ from $\Omega_{(B,b)}$ (line 11); we then expand X and the pattern $P(X)$ so that the premise ω can be satisfied (line 13), until either $(X, P(X))$ can no longer satisfy ω (success $= false$) (line 14), or each $b_i \prec_B^v b$ in $U_{(B,b)}$ is covered by such a rule ϕ (success $= true$); in the latter case, we add the rule $(X, P(X)) \rightarrow (B,b)$ to Π (line 15). Note that $|X| \leq |R|$.

The procedure is in $O((|\Sigma| + |\Gamma|)|I_t|^2 + |I_t|^3)$ time. Indeed, the cost of step (1) is bounded by $O(|\Gamma|)$; and for step (2), since $U_{(B,b)}$'s are disjoint, $\Omega_{(B,b)}$'s partition $\Omega(S_e)$, and moreover, each ϕ in $\Omega(S_e)$ is used at most once, it takes at most $O((|\Sigma| + |\Gamma|)|I_t|^2 + |I_t|^3)$ time.

5.5.3.2.3 CompGraph**:** Given a set of derivation rules, procedure CompGraph generates their compatibility graph $G(N,E)$ (see Example 5.5.5 for a running example).

More specifically, CompGraph is presented in Fig. 5.10. It takes a set Π of derivation rules as input. It constructs and returns a compatibility graph for Π. The procedure works as follows. It first initializes a compatibility graph (line 1). It then generates a node for each derivation rule (line 2). The edges are then added (lines 3-6). For any two distinct nodes, if their associated rules are compatible (line 5, see the definition of compatibility graphs given earlier), an edge is added to connect these two nodes (line 6). It terminates and returns a compatibility graph G (line 7).

It is readily to verify that the procedure takes at most $O(|\Pi|^2)$ time, where $|\Pi|$ is no larger than $|R||I_t|$.

5.5.3.2.4 MaxClique**:** Given a compatibility graph $G(N,E)$, this procedure computes a maximum clique $\mathcal{C}$ of $G(N,E)$. While it is intractable to find a maximum clique, several tools have been developed for computing maximum cliques, with a good approximation bound (*e.g.,* [Fei05]). We employ one of these tools as MaxClique.

5.5.3.2.5 GetSug**:** Given a specification S_e of an entity, a set $\Omega(S_e)$ of instance constraints, a clique $\mathcal{C}$, and $\mathcal{V}_{\mathcal{B}}$, the procedure computes a suggestion as output. As shown in Examples 5.5.6 and 5.5.7, the clique returned by MaxClique represents a suggestion, but the suggestion may contain conflicts. This procedure is to convert the the clique to a suggestion, and revise it in the presence of conflicts by invoking a weighted MaxSat-solver.

More specifically, GetSug is given in Fig. 5.11. It works as follows. It first

```
Procedure GetSug

Input: A specification $S_e$, instance constraints $\Omega(S_e)$,
       a clique $\mathcal{C}$, and $\mathcal{V}_{\mathcal{B}}$.
Output: A set $\mathcal{A}$ of attributes as suggestion.

1.  $\mathcal{A}' := R; P'[\mathcal{A}'] := nil$;
2.  for each $(X, P(X)) \to (B, b) \in \mathcal{C}$ do
3.      $\mathcal{A}' := \mathcal{A}' \setminus \{B\}$
4.      $P'[X] := P(X)$;
5.  instantiate $P'$ and convert $P'$ to CNF $\Phi'$;
6.  $\Phi_u := \Phi' \cup \Phi(S_e)$;
    /* each clause in $\Phi'$ is assigned with a weight of 1.0,
    and each clause in $\Phi(S_e)$ is assigned with a weight of $+\infty$.*/
7.  $\Phi_s :=$ Weighted-MAXSAT ($\Phi_u$);
8.  convert $\Phi_s$ to $P''$;
9.  $\mathcal{A} := R \setminus \mathcal{V}_{\mathcal{B}}$;
10. for each $(X, P(X)) \to (B, b) \in \mathcal{C}$ do
11.     if $P(X) \neq P''[X]$ then remove $(X, P(X)) \to (B, b)$ from $\mathcal{C}$;
12.     else $\mathcal{A} := \mathcal{A} \setminus \{B\}$;
13. return $\mathcal{A}$;
```

Figure 5.11: Procedure GetSug

identifies the required attributes $\mathcal{A}'$ and pattern $P'[\mathcal{A}]$ by applying derivation rules in $\mathcal{C}$ (lines 1-4). It then converts P' to a CNF Φ', along the same line as procedures Instantiation and ConvertToCNF given earlier (line 5). Since Φ' may have conflicts with the $\Phi(S_e)$, it invokes a weighted MAXSAT-solver to minimally revise Φ' such that $\Phi' \cup \Phi(S_e)$ is satisfiable (lines 6-7). It then finds the subset of $\mathcal{C}$ corresponding to the revised Φ', which has no conflicts with S_e (lines 8-12). It also derives a set $\mathcal{A}$ of attributes from the subset of $\mathcal{C}$ (line 12; see Example 5.5.6). Finally, it returns $\mathcal{A}$ (line 13). Recall that $\mathsf{V}(A)$ is computed by procedure DeriveVR given earlier. Note that the input to the MaxSat-solver is no larger than $|R|^2|I_t|^2$. Moreover, there are efficient MaxSat-solvers available, with a reasonable approximation bound [SK04].

Correctness. Algorithm Suggest guarantees to generate a suggestion $(\mathcal{A}, \mathsf{V}(\mathcal{A}))$. Indeed, (1) the clique $\mathcal{C}'$ revised by MaxSat has no conflicts with S_e, and thus $\mathcal{C}'$ and S_e warrant to have a valid completion I_t^c. Let $t_c = \mathsf{LST}(I_t^c)$. If $\mathsf{V}(\mathcal{A})$ are validated for

$\mathcal{A}$, then t_c must be the *true value* $\mathsf{T}(S_e)$ of S_e, since $t_c[\mathcal{B}] = \mathsf{V}_{\mathcal{B}}$ remains unchanged for all valid completions of S_e, and $t_c[\mathcal{A}']$ is uniquely determined by $t_c[\mathcal{A}]$ and $\mathsf{V}_{\mathcal{B}}$ by the construction. (2) All possible true values for $\mathcal{A}$ from their active domains are already included in $\mathsf{V}(\mathcal{A})$.

5.6 Experimental Study

We conducted experiments using both real-life and synthetic data. We evaluated the accuracy and scalability of (1) IsValid for validating a specification, (2) DeduceOrder for deducing true values, (3) Suggest for computing suggestions, and (4) the overall performance of conflict resolution supporting (1-3) above. Note that IsValid and DeduceOrder are useful in their own right, since users may want to check their specifications and infer true values outside the interaction framework.

Experimental data. We used two real-life datasets (NBA and CAREER) and synthetic data (Syn). Constraints were discovered using profiling algorithms [CM08, FGLX11], and examined manually. Timestamps for the datasets were either missing (for CAREER and Syn) or incomplete (NBA). We assumed *empty currency orders* in all the experiments even when partial timestamps were given. The available (incomplete) timestamps were used for designing currency constraints and for evaluating the quality of the derived true values.

NBA *player statistics*. This dataset was retrieved from the following sites: (1) *http://databasebasketball.com/*, (2) *http://www.infochimps.com/marketplace*, and (3) *http://en.wikipedia.org/wiki/List_of_National_Basketball _Association_arenas*. It consists of three tables: (a) Player (from sources 1 and 3) contains information about players, identified by player id (pid). (b) Stat (from 1) includes the statistics of these players from the 2005/2006 season to the 2010/2011 season. (c) Arenas (from 3) records the historical team names and arenas of each team. We created a table, referred to as NBA, by first joining Player and Stat via *equi-join* on the pid attribute, and then joining Arenas via *equi-join* on the team attribute. The NBA table consists of 19573 tuples for 760 entities (*i.e.,* players). Its schema is (pid, name, true name, team, league, tname, points, poss, allpoints, min, arena, opened, capacity, city). When producing the NBA table we took care of the attributes containing multiple values for a player, *e.g.,* multiple teams for the same player, and multiple teams for one arena. We ensure that only one attribute value (*e.g.,* team) appears in any tuple. Only data from (1) and (3) carries (partial) timestamps. Therefore, the true values of entities in the NBA table *cannot* be directly derived when putting (1), (2) and (3) together.

The number of tuples pertaining to an entity ranges from 2 to 136, about 27 in average. We consider *entity instances*, *i.e.,* tuples referring to the same entity, which are *much smaller* than a database.

We found 54 currency constraints: 15 for team names (tname) as shown by φ_1 below; 32 for arena, similar to φ_2; and 4 (resp. 3) for attribute allpoints that were scored since 2005 (resp. arena), similar to φ_3 (resp. φ_4), where B ranges over points, poss, min and tname (resp. opened, capacity and years). We deduced 58 constant CFDs, *e.g.,* ψ_1 below. Note that some rules are derived automatically, while the others are designed manually based on the semantics of the data.

φ_1: $\forall t_1, t_2\ (t_1[\mathsf{tname}] = \text{“New Orleans Jazz”}$
$\quad\quad \wedge\ t_2[\mathsf{tname}] = \text{“Utah Jazz”} \rightarrow t_1 \prec_{\mathsf{tname}} t_2)$;

φ_2: $\forall t_1, t_2\ (t_1[\mathsf{arena}] = \text{“Long Beach Arena”}$
$\quad\quad \wedge\ t_2[\mathsf{arena}] = \text{“Staples Center”} \rightarrow t_1 \prec_{\mathsf{arena}} t_2)$;

φ_3: $\forall t_1, t_2\ (t_1[\mathsf{allpoints}] < t_2[\mathsf{allpoints}] \wedge t_1[B] \neq t_2[B]$
$\quad\quad\quad\quad \rightarrow t_1 \prec_{\mathsf{B}} t_2)$

φ_4: $\forall t_1, t_2\ (t_1 \prec_{\mathsf{arena}} t_2 \wedge t_1[B] \neq t_2[B] \rightarrow t_1 \prec_{\mathsf{B}} t_2)$

ψ_1: $(\mathsf{arena} = \text{“United Center”} \rightarrow \mathsf{city} = \text{“Chicago, Illinois”})$

(2) CAREER. This data set was retrieved *as is* from the source *http://www.cs.purdue.edu/commugrate/data/citeseer*. Its schema is (first name, last name, affiliation, city, country). We chose 65 persons from the dataset, and for each one, we collected all of his/her publications, one tuple for each. *No reliable timestamps* were available for this dataset. The number of tuples pertaining to an entity ranges from 2 to 175, about 32 in average.

We derived 503 currency constraints: if two papers A and B are by the same person and A cites B, then the affiliation and address (city and country) used in paper A are more current than those used in paper B. We also found a single CFD of the form: (affiliation → city, country), but with 347 patterns with different constants.

The constraints for each dataset (NBA and CAREER) have essentially *the same form*, and *only differ* in their constants. Indeed, we find that the number of constraints with different forms is rather *small* in practice.

(3) Syn *data*. The synthetic data adheres to the schema given in Table 5.2. We found

983 currency constraints (of *the same form* but with distinct constant values for status, job and kid) and a single CFD AC → city with 1000 patterns (counted as distinct constant CFDs), similar to those in Table 5.3. The data generator used two parameters: n denotes the number of entities, and s is the size of *entity instances* (the number of tuples pertaining to an entity). For each entity, it first generated a true value t_c, and then produced a set E of tuples that have conflicts but do not violate the currency constraints; we treated $E \setminus \{t_c\}$ as the entity instance. We generated n = 10k entities, with s from 1 to 10k. We used *empty* currency orders here.

Algorithms. We implemented the following algorithms in *C++*: (a) IsValid (Section 5.5.1): it calls MiniSat [GT04] as the SAT-solver; (b) DeduceOrder and NaiveDeduce, where NaiveDeduce repeatedly invokes MiniSat [GT04], as described in Section 5.5.2; and (c) Suggest: it uses MaxClique [Fei05] to find a maximal clique, and MaxSat-solver [SK04] to derive a suggestion (Section 5.5.3). We simulated user interactions by providing true values for suggested attributes, some with new values, *i.e.,* values not in the active domain. We also implemented (d) Pick, a traditional method that randomly takes a value [BN08]; to favor Pick, we picked a value from those that are not less current than any other values, based on currency constraints $\forall t_1, t_2(\omega \rightarrow t_1 \prec_A t_2)$ in which ω is a conjunction of comparison predicates only, *e.g.,* φ_1–φ_3 above.

Accuracy. To measure the quality of suggestions, we used F-measure (*http://en.wikipedia.org/wiki/F-measure*):

$$\text{F-measure} = 2 \cdot (\text{recall} \cdot \text{precision})/(\text{recall} + \text{precision}).$$

Here precision is the ratio of the number of values correctly deduced to the total number of values deduced; and recall is the ratio of the number of values correctly deduced to the total number of attributes with conflicts or stale values.

All experiments were conducted on a Linux machine with a 3.0GHz Intel CPU and 4GB of Memory. Each experiment was repeated 5 times, and the average is reported here.

Experimental results. We next present our findings. Due to the small size of the CAREER data for each entity, experiments conducted on it took typically less than 10 milliseconds (ms). Hence we do not report its result in the efficiency study.

Exp-1: Validity checking. We first evaluated the scalability of IsValid. The average time taken by entity instances of various sizes is reported in Fig. 5.12(a), where the lower x-axis shows the sizes of NBA, and the upper x-axis is for Syn data. The results show that IsValid suffices to validate specifications of a reasonably large size. For example, it took 220 ms for NBA entity instances of 109-135 tuples and 112 constraints, with 14 attributes in each tuple. For Syn, it took an average of 4.7 seconds on entities of 8k-10k tuples and 1983 constraints. We also find IsValid accurate: specifications reported (in)valid are indeed (in)valid.

Exp-2: Deducing true values. We next evaluated the performance of algorithms DeduceOrder and NaiveDeduce. The results on both NBA and Syn data are reported in Fig. 5.12(b), which tell us the following: (a) DeduceOrder scales well with the size of entity instances, and (b) DeduceOrder substantially outperforms NaiveDeduce on both datasets, for reasons given in Section 5.5.2. Indeed, DeduceOrder took 51 ms on NBA entity instances with 109-135 tuples, and 914 ms on Syn entities of 8k-10k tuples; in contrast, NaiveDeduce spent 13585 ms and over 20 minutes (hence not shown in Fig. 5.12(b)) on the same datasets, respectively.

We also find that DeduceOrder derived as many true values as NaiveDeduce on both datasets (not shown). This tells us that DeduceOrder can efficiently deduce true values on large entity instances without compromising the accuracy of the true values found.

Exp-3: Suggestions for user interactions. We evaluated the accuracy of suggestions generated from currency constraints Σ and CFDs Γ put together. The results on NBA, CAREER and Syn are given in Figures 5.12(e), 5.12(i) and 5.12(m), respectively, where the x-axis indicates the rounds of interactions, and the y-axis is the percentage of true attribute values deduced.

These results tell us the following. (a) Few rounds of interactions are needed to find all the true attribute values for an entity: at most 2, 2 and 3 rounds for NBA, CAREER and Syn data, respectively. (b) A large part of true values can be *automatically deduced* by means of currency and consistency inferences: 35%, 78% and 22% of true values are identified from $\Sigma + \Gamma$ *without user interaction*, as indicated by the 0-interaction in Figures 5.12(e), 5.12(i) and 5.12(m), respectively.

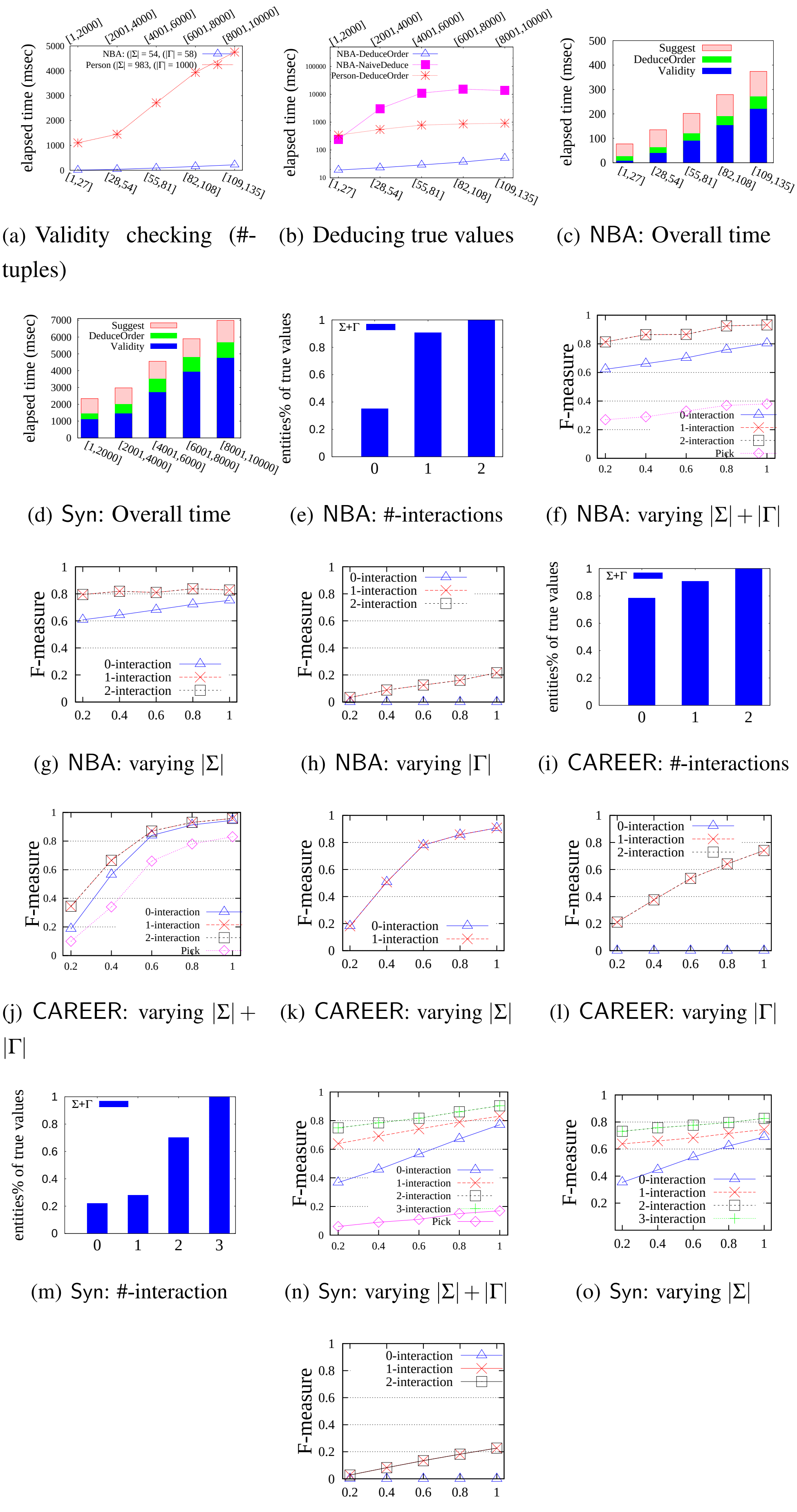
Impact of $|\Sigma|$ *and* $|\Gamma|$. To be more precise when evaluating the accuracy, we use F-measure, which combines precision and recall, and take the cases of using $|\Gamma|$ only or $|\Sigma|$ only into consideration. Figures 5.12(f)–5.12(h), 5.12(j)–5.12(l) and 5.12(n)–5.12(p) show the results for NBA, CAREER and Syn, respectively, when varying both $|\Sigma|$ and $|\Gamma|$, $|\Sigma|$ only, and varying $|\Gamma|$ alone, respectively. The x-axis shows the percentage of Σ or Γ used, and the y-axis shows the corresponding F-measure values.

These results tell us the following. (a) As shown in Figures 5.12(f), 5.12(j) and 5.12(n), our method substantially outperforms the traditional method Pick, by 201% in average on all datasets, even when we favor Pick by allowing it to capitalize on currency orders. This verifies that data currency and consistency can significantly improve the accuracy of conflict resolution. (b) When Σ and Γ are taken together, the F-measure value is up to 0.930 for NBA (Fig. 5.12(f), the top right point), 0.958 for CAREER (Fig. 5.12(j)), and 0.903 for Syn (Fig. 5.12(n)), in contrast to 0.830 in Fig. 5.12(g), 0.907 in Fig. 5.12(k), and 0.826 in Fig. 5.12(o), respectively, when Σ is used alone, and as opposed to 0.210 in Fig. 5.12(h), 0.741 in Fig. 5.12(l), and 0.234 in Fig. 5.12(p), respectively, with Γ only. These further verify that the inferences of data currency and consistency should be *unified* instead of taking separately. (c) The more currency constraints and/or CFDs are available, the higher the F-measure is, as expected. (d) The two curves for the 2- and 1-interaction overlap in Figures 5.12(f)–5.12(h) for NBA, 2- and 1-interaction in Figures 5.12(j)–5.12(l) for CAREER, and 3- and 2-interaction in Figures 5.12(n)–5.12(p) for Syn. These indicate that the user interactions are needed to provide true values for those attributes that we do not have enough information to deduce their true values.

Exp-4: Efficiency. The overall performance for resolving conflicts in the NBA (resp. Syn) data is reported in Fig. 5.12(c) (resp. Fig. 5.12(d)). Each bar is divided into the elapsed time taken by (a) validity checking, (b) true value deducing, and (c) suggestion generating, including computing maximal cliques and running MaxSat. The result shows that conflict resolution can be conducted efficiently in practice, *e.g.,* each round of interactions for NBA took 380 ms. Here validating specifications takes most time, dominated by the cost of SAT-solver, while deducing true values takes the least time.

Summary. From the experimental results we find the following. (a) Conflict resolution by reasoning about data currency and consistency substantially outperforms the traditional method Pick, by 201%. (b) It is more effective to unify the inferences of data currency and consistency than treating them independently. Indeed, when Σ and Γ are taken together, the F-measure improves over Σ only and Γ only by 11% and 236%, respectively. (c) Our conflict resolution method is efficient: it takes less than 0.5 second on the real-life datasets even with interactions. (d) Our method scales well with the size of entities and the number of constraints. Indeed, it takes an average of 7 seconds to resolve conflicts in Syn entity instances of 8k-10k tuples, with 1983 constraints. (e)

At most 2-3 rounds of interactions are needed for all datasets.



(a) Validity checking (#-tuples)

(b) Deducing true values

(c) NBA: Overall time

(d) Syn: Overall time

(e) NBA: #-interactions

(f) NBA: varying $|\Sigma|+|\Gamma|$

(g) NBA: varying $|\Sigma|$

(h) NBA: varying $|\Gamma|$

(i) CAREER: #-interactions

(j) CAREER: varying $|\Sigma|+|\Gamma|$

(k) CAREER: varying $|\Sigma|$

(l) CAREER: varying $|\Gamma|$

(m) Syn: #-interaction

(n) Syn: varying $|\Sigma|+|\Gamma|$

(o) Syn: varying $|\Sigma|$

(p) Syn: varying $|\Gamma|$

Figure 5.12: Experimental results

Chapter 6

Determining the Relative Accuracy of Attributes

6.1 Introduction

In this chapter, we next report our study on (relative) data accuracy, one of the central issues.

Given a set I_e of tuples pertaining to the same entity e, data accuracy aims to find the most accurate values for e. More specifically, it is to compute a tuple t_e, referred to the *target tuple for e from* I_e, such that for each attribute A of e, $t_e[A]$ is a value in I_e that is closest to the *true A-value* of e.

Although important, data accuracy has not been well studied by and large. Prior work on data quality has typically focused on other issues such as data consistency [BS06, FG12]. Consistency refers to the validity and integrity of the data. Although data accuracy and consistency are connected, the two are *quite different*. Indeed, a database D may be consistent, but the values in D may still be inaccurate.

	FN	MN	LN	rnds	totalPts	J#	league	team	arena
t_1:	MJ	null	null	16	424	45	NBA	Chicago	Chicago Stadium
t_2:	Michael	null	Jordan	27	772	23	NBA	Chicago Bulls	United Center
t_3:	Michael	null	Jordan	1	19	45	NBA	Chicago Bulls	United Center
t_4:	Michael	Jeffrey	Jordan	127	51	45	SL	Birmingham Barons	Regions Park

Table 6.1: Entity instance stat for Michael Jordon in the 1994-95 season

Example 6.1.1: Consider relation stat given in Table 6.1, which collects performance

	FN	LN	league	season	team
s_1:	Michael	Jordan	NBA	1994-95	Chicago Bulls
s_2:	David	Robinson	NBA	1994-95	San Antonio Spurs
s_3:	Michael	Jordan	NBA	2001-02	Washington Wizards

Table 6.2: Master data nba

φ_1: $\forall t_1, t_2 \in \mathsf{stat}\ (t_1[\mathsf{league}] = t_2[\mathsf{league}] \wedge t_1[\mathsf{rnds}] < t_2[\mathsf{rnds}] \rightarrow t_1 \preceq_{\mathsf{rnds}} t_2)$

φ_2 : $\forall t_1, t_2 \in \mathsf{stat}\ (t_1 \prec_{\mathsf{rnds}} t_2 \rightarrow t_1 \preceq_{\mathsf{J\#}} t_2)$

φ_3: $\forall t_1, t_2 \in \mathsf{stat}\ (t_1 \prec_{\mathsf{rnds}} t_2 \rightarrow t_1 \preceq_{\mathsf{totalPts}} t_2)$

φ_4 : $\forall t_1, t_2 \in \mathsf{stat}\ (t_1 \prec_{\mathsf{league}} t_2 \rightarrow t_1 \preceq_{\mathsf{rnds}} t_2)$

φ_5 : $\forall t_1, t_2 \in \mathsf{stat}\ (t_1 \prec_{\mathsf{MN}} t_2 \rightarrow t_1 \preceq_{\mathsf{FN}} t_2)$

φ_6 : $\forall t_m \in \mathsf{nba}(t_m[\mathsf{FN}, \mathsf{LN}] = t_e[\mathsf{FN}, \mathsf{LN}] \wedge t_m[\mathsf{season}] = \text{“1994-95”} \rightarrow t_e[\mathsf{league}, \mathsf{team}] = t_m[\mathsf{league}, \mathsf{team}])$

Table 6.3: Accuracy rules

statistics of Michael Jordan (Fig. 6.1) in the season of 1994-95, when Michael played for a baseball team Birmingham Barons in the Southern League (SL) in 1994, followed by his return to NBA, playing 27 games for Chicago Bulls in 1995. Each tuple in stat specifies the name (FN, MN, LN), performance (total points totalPts after rnds rounds played), jersey number J#, league, team and arena.

We want to find the target tuple for Michael from stat, consisting of the most accurate values for all the attributes at the end of 1994-95 NBA season. However, there are multiple values in stat for some attributes. For instance, we do not know whether 45 is more accurate than 23 for his J#.

The stat data is *consistent*. Indeed, constraints specifying its consistency include (a) functional dependency (FD [AHV95]): [FN, MN, LN, league, rnds → totalPts], *i.e.,* player, rnds and league uniquely determine totalPts, and (b) conditional functional dependency (CFD [BFG$^+$07]): [team = “Chicago Bulls” → arena = “United Center”], asserting that if team is Chicago Bulls, then arena must be United Center. While all tuples in stat satisfy these constraints and are hence *consistent*, most of the data values in stat are, however, *not accurate*. □

Not all is lost. By using master data, one is able to identify accurate values for *some* attributes. For example, a master relation nba is given in Table 6.2, in which a tuple specifies the FN, LN, league, season, and team of an NBA player. Then tuple s_1 in Table 6.2 tells us that in the 1994-95 season, Michael played for Chicago Bulls

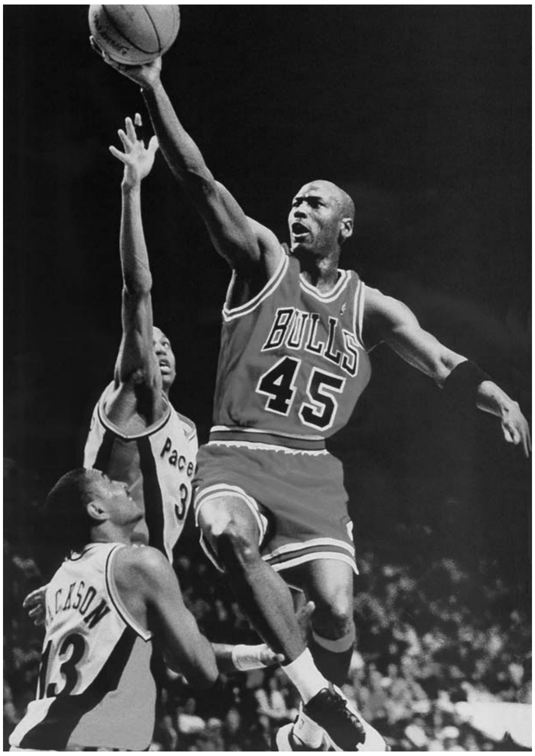


Figure 6.1: MJ's return

in NBA. Thus t_1 is *more accurate* than t_4 in *attribute* league (resp. team), denoted by $t_6 \prec_{\mathsf{league}} t_1$ (resp. $t_4 \prec_{\mathsf{team}} t_1$).

To find the *relative accuracy* of those attributes *not presented* in the master data is much more challenging. That is, given tuples t_1, t_2 and an attribute A, we want to find whether $t_1 \prec_A t_2$ when A is *not* covered by master data, such as J# in relation stat of Table 6.1. This is hard, but not hopeless. From the semantics of the data, one can discover accuracy rules (ARs), which tell us whether one tuple is more accurate than another in certain attributes.

Example 6.1.2: An analysis of the semantics of the stat data yields the ARs given in Table 6.1. Based on these rules, we can deduce relative accuracy as follows.

(1) We know that in a season, the number of rounds (rnds) monotonically increases (up to an bound). Hence for tuples t and t' referring to the same league, if $t[\mathsf{rnds}] < t'[\mathsf{rnds}]$, then $t \prec_{\mathsf{rnds}} t'$, *i.e.,* $t'[\mathsf{rnds}]$ is more current (and thus more accurate) than $t[\mathsf{rnds}]$. This is expressed as rule φ_1 in Table 6.1. From φ_1 we can deduce that $t_i \prec_{\mathsf{rnds}} t_2$ for $i \in [1,3]$.

(2) For tuples t and t', if t' is more accurate than t in rnds, then $t \preceq_{\mathsf{J\#}} t'$, denoting either $t[\mathsf{J\#}] = t'[\mathsf{J\#}]$ or $t \prec_{\mathsf{J\#}} t'$; similarly for totalPts. That is, if t' is more accurate (current)

than t in attribute rnds, then so are its *correlated attributes* $t'[\mathsf{J\#}]$ and $t'[\mathsf{totalPts}]$. These are expressed as rules φ_2 and φ_3 in Table 6.1, respectively. From these ARs and (1) we find that $t_2[\mathsf{totalPts}] = 772$ and $t_2[\mathsf{J\#}] = 23$ are more accurate than $t_i[\mathsf{totalPts}]$ and $t_i[\mathsf{J\#}]$, respectively, for $i \in [1,3]$.

(3) We know that Michael ended up in NBA in the 1994-95 season. Moreover, if t' is more accurate than t in league, then so is t' in attributes rnds, $\mathsf{totalPts}$, $\mathsf{J\#}$ and arena. These can also be expressed as ARs, *e.g.,* φ_4. These tell us that t_i is more accurate than t_4 in these attributes, for $i \in [1,3]$.

(4) For tuples t and t', if $t[\mathsf{A}]$ is null but $t'[\mathsf{A}]$ is not, then t' is more accurate than t in attribute A. This can also be expressed as an AR (not shown in Table 6.1). Moreover, if t' is more accurate than t in MN, then so is t' in the *correlated attribute* FN, as $t'[\mathsf{MN}]$ and $t'[\mathsf{FN}]$ typically *come together*. This is expressed as φ_5 in Table 6.1. These tell us that $t_4[\mathsf{MN}]$="Jeffrey" is most accurate in MN and $t_1 \prec_{\mathsf{FN}} t_4$.

(5) As remarked earlier, we can use master data to find the most accurate values for certain attributes. This is shown as AR φ_6 in Table 6.1. It asserts that if there exists a master tuple $t_m \in \mathsf{nba}$ such that $t_m[\mathsf{FN},\mathsf{LN}] = t_e[\mathsf{FN},\mathsf{LN}]$ and $t_m[\mathsf{season}]$ = "1994-95", then $t_e[\mathsf{league},\mathsf{team}]$ should take the value of $t_m[\mathsf{league},\mathsf{team}]$. Here t_e is the target tuple in which attributes $t_e[\mathsf{FN},\mathsf{LN}]$ already find their most accurate values.

Putting these together, we can deduce the relative accuracy of attributes and better still, a large part of the target tuple t_e. Indeed, the values of t_e in FN, MN, LN, rnds, $\mathsf{totalPts}$, league, team are found to be Michael, Jeffrey, Jordan, 27, 772, 23, NBA, and Chicago Bulls, respectively. □

This example tells us that even *in the absence of true values* of an entity, one can still find a large part of the target tuple for the entity, by taking both accuracy rules and available master data. This, however, requires an inference system and efficient algorithms for deducing the relative accuracy of attributes whenever possible.

We contend that ARs, master data and inference algorithms yield a promising approach to determining relative accuracy. As shown in Example 6.1.2, while master data is helpful, it is *not* a must for the analysis of relative accuracy. In the absence of complete master data, we can still deduce true values for critical attributes based on accuracy rules and inference, as will be verified by our experimental study.

Related works. There has been a host of work on data quality issues such as data consistency, data currency, information completeness and entity resolution (see [BS06, FG12] for recent surveys). While data accuracy has long been advocated [BS06, FLM09, NJE$^+$09], the prior work has mostly focused on metrics for accuracy measurement; we are not aware of any formal treatment of relative accuracy in the absence of true values.

Rules and master data have been used in repairing data (data consistency) [BKL11, FLM$^+$11a] and specifying relative information completeness [FG12]. This work differs from the prior work in the following. (1) ARs are quite different from the dependencies used for specifying consistency and completeness. As a result, the termination problem for rules of [FLM$^+$11a], for instance, is PSPACE-complete, while our inference (chase) process always terminates. (2) Data repairing and information completeness consider problems different from those studied here. (3) We give an operational semantics for ARs in terms of chase [AHV95]. In contrast, chase was not considered in the prior work except [BKL11]. While [BKL11] adapted chase for data repairing [BKL11] based on matching dependencies, it studied neither how to deduce relative accuracy, nor the complexity of determining whether a chase process is Church-Rosser.

Also related is prior work on truth discovery from data sources [BCMP10, FGTY13, GAMS10, WM11, YHY08, DBES09b, Wid05, ZRGH12]. Those approaches include (i) dependencies on sources to detect copy relationships and identify reliable sources [DBES09b]; (ii) employing lineage and probabilistic [Wid05]; (iii) vote counting and probabilistic analysis based on the trustworthiness of data sources [BCMP10, GAMS10, WM11, YHY08, ZRGH12]. In contrast, we deduce relative accuracy following a logical approach based on ARs and master data, without assuming knowledge about data sources. Our method is *complementary to* the prior approaches for truth discovery, and can be combined with them by deducing trust in attributes with ARs in truth discovery (see Sections 6.3 and 6.7 for details).

Closer to this work is [FGTY13], on conflict resolution by reasoning about data consistency and currency. It used partial orders, currency constraints and constant CFDs [BFG$^+$07]. This work differs from [FGTY13] in the following. (1) We study relative data accuracy rather than conflict resolution. This said, currency orders and constant CFDs can be expressed as ARs and hence, our techniques can also be used in data fusion [BN08]. (2) We use ARs for actions in a chase process, as opposed to static dependencies of [FGTY13]. (3) Our approach is quite different from [FGTY13]. We infer accuracy via chase, use available master data to improve accuracy, and provide algorithms to compute top-k target tuples. These were not studied in [FGTY13].

Our algorithms for computing top-k target tuples are related to top-k query answering, which aims to retrieve top-k tuples from query result, ranked by a monotone scoring function [IBS08, FLN03]. One of our top-k algorithms extends algorithms for top-k rank join queries [SP08, IAE04] by *embedding* score computation in top-k selection, rather than assuming that the scores are already given, and by additionally checking whether selected tuples observe ARs. We also provide a new algorithm that *does not* require the input to be ranked; it is instance optimal *w.r.t.* the number of visits to the data, and can be used to *compute rank joins on unranked lists*.

Organization. The rest of the chapter is organized as follows. We formally propose a class of accuracy rules (ARs) to specify data accuracy, and a chase-like procedure in Section 6.2. We study the fundamental problems of ARs in Section 6.3, and introduce a framework to derive data accuracy in Section 6.4. Algorithm for checking Church-Rosser property is given in Section 6.5, and practical top-k algorithms underlying the framework are developed in Section 6.6. An experimental study is reported in Section 6.7

6.2 A Model for Relative Accuracy

We next present a model for determining relative accuracy. We first define ARs (Section 6.2.1), and then introduce a chase procedure for deducing relative accuracy (Section 6.2.2),

6.2.1 Rules for Specifying Relative Accuracy

Relative accuracy. Consider a relation schema $R = (A_1, \ldots, A_n)$, where the domain of attribute A_i is $\mathsf{dom}(A_i)$. We consider an *entity instance* I_e of R, which is a set of tuples pertaining to the *same* real-world entity e. Such an I_e is identified by entity resolution techniques [EIV07, NH10], and is typically *much smaller* than a database instance in practice.

The problem of relative accuracy is to determine, given an attribute A_i of R and tuples $t_1, t_2 \in I_e$, whether t_2 is *more accurate than* t_1 *in attribute* A_i, denoted by $t_1 \prec_{A_i} t_2$.

More specifically, for each attribute A_i of I_e, $\prec_{A_i}$ is a *strict partial order* defined on the A_i attribute values in I_e. That is, $\prec_{A_i}$ is a binary relation that is irreflexive and transitive, and thus asymmetric. Initially, $\prec_{A_i}$ is *empty* for all $i \in [1, n]$, and we want to populate $\prec_{A_i}$ by deducing relative accuracy with accuracy rules. We also use $t_1 \preceq_{A_i} t_2$

to denote either $t_1[A_i] = t_2[A_i]$ or $t_1 \prec_{A_i} t_2$. Note that $\preceq_{A_i}$ is a partial order, referred to as the *accuracy order on attribute A_i*.

Ultimately we want to find a tuple t_e for I_e, referred to as the *target tuple for e from I_e* such that for each attribute A_i of R and all tuples $t \in I_e$, $t \preceq_{A_i} t_e$. Intuitively, t_e is a *new tuple* composed of the most accurate value of attribute A_i for all $i \in [1,n]$. It is easy to verify that if t_e exists, then it is *unique*. Note that I_e may not have enough information for us to deduce a complete t_e. If so, $t_e[A_i] = \text{null}$ for some A_i, and we refer to t_e as an *incomplete target tuple* of I_e.

Accuracy rules (ARs). There are two forms of ARs. The first one is defined on tuples $t_1, t_2 \in I_e$ to deduce whether $t_1 \preceq_{A_i} t_2$, *i.e.,* their relative accuracy in an attribute A_i:

$$\varphi = \forall t_1, t_2\ (R(t_1) \wedge R(t_2) \wedge \omega \rightarrow t_1\ \preceq_{A_i}\ t_2) \qquad (1)$$

where ω is a conjunction of predicates of the form: (a) $t_1[A_l]$ op $t_2[A_l]$, where op is one of the comparison operators $=, \neq, >, <, \leq, \geq$; or (b) $t_i[A_l]$ op c for $i \in [1,2]$, where c is a constant or $t_e[A_l]$; or (c) $t_1 \prec_{A_l} t_2$ or $t_1 \preceq_{A_l} t_2$. We refer to ω as LHS(φ) and $t_1 \preceq_{A_i} t_2$ as RHS(φ).

We denote by $(t_1, t_2) \models \omega$ if t_1 and t_2 satisfy the predicates in ω following the standard semantics of first-order logic. Intuitively, if $(t_1, t_2) \models \omega$, then $t_1 \prec_{A_i} t_2$ or $t_1 \preceq_{A_i} t_2$.

The second form of ARs is defined on (t_e, I_m), where t_e is the *target tuple template*, and I_m is an available master relation of schema R_m [RW08]. Note that R_m may not cover all the attributes of R. This form of ARs extends t_e by extracting accurate values from master relation I_m, as follows:

$$\varphi' = \forall t_m\ (R_m(t_m) \wedge \omega \rightarrow t_e[A_i] = t_m[B]) \qquad (2)$$

Here ω is a conjunction of predicates of the form $t_e[A_l] = c$ or $t_e[A_l] = t_m[B']$, where c is a constant and B' is an attribute of R_m. Intuitively, if t_e matches a master tuple t_m in I_m as specified by ω, then $t_e[A_i]$ is *instantiated* by taking the value of $t_m[B]$. We refer to ω as LHS(φ') and $t_e[A_i] = t_m[B]$ as RHS(φ). We write $(t_e, t_m) \models \omega$ if t_e and t_m satisfy ω.

Example 6.2.1: Recall the entity instance stat of Table 6.1, and master relation nba of Table 6.2. Then their ARs include φ_1–φ_6 given in Table 6.1. These ARs demonstrate how we can derive relative accuracy, in terms of (a) constants, built-in predicates and the semantics of the data such as φ_1, (b) data currency, *e.g.,* φ_2 and φ_3, (c) *co-existence* of attributes and known accuracy orders, such as φ_4 and φ_5, and (d) available master data such as φ_6. Additional ARs for stat include:

φ_7: $\forall t_1, t_2 \in \mathsf{stat}\ \big(t_1[A] = \mathsf{null} \wedge t_2[A] \neq \mathsf{null} \rightarrow t_1 \preceq_A t_2\big)$
φ_8: $\forall t_1, t_2 \in \mathsf{stat}\ \big(t_2[A] = t_e[A] \wedge t_e[A] \neq \mathsf{null} \rightarrow t_1 \preceq_A t_2\big)$
φ_9: $\forall t_1, t_2 \in \mathsf{stat}\ \big(t_1[A] = t_2[A] \rightarrow t_1 \preceq_A t_2\big)$
φ_{10}: $\forall t_1, t_2 \in \mathsf{stat}\ \big(t_1 \prec_{\mathsf{MN}} t_2 \rightarrow t_1 \preceq_{\mathsf{LN}} t_2\big)$
φ_{11}: $\forall t_1, t_2 \in \mathsf{stat}\ \big(t_1 \prec_{\mathsf{team}} t_2 \rightarrow t_1 \preceq_{\mathsf{arena}} t_2\big)$

Here φ_7–φ_9 are defined on all the attributes A of stat. Rule φ_7 says that the null value has the lowest accuracy; φ_8 asserts that if the target attribute $t_e[A]$ is defined, then it has the highest accuracy among all A-attribute values in I_e; and φ_9 says that for all t_1 and t_2, if $t_1[A] = t_2[A]$, then $t_1 \preceq_A t_2$. Rules φ_7–φ_9 are "axioms" that are included in any set of ARs. ARs φ_{10} and φ_{11} deduce accuracy from *correlated attributes* (*e.g.,* φ_{10}, if a tuple has a more accurate MN, then so does LN since the two attributes often come together).

Note that $t_1 \prec_A t_2$ iff $t_1 \preceq_A t_2$ and $t_1[A] \neq t_2[A]$. Hence when $\preceq_A$ is computed, we *can derive* $\prec_A$ *from* $\preceq_A$. □

Remark. Constant CFDs [BFG+07] developed for detecting data inconsistencies can be expressed as ARs. As an example, consider the CFD ψ given in Example 6.1.1: [team = "Chicago Bulls" → arena = "United Center"]. We can create a master relation of schema R_m with a tuple (team = "Chicago Bulls", arena = "United Center"), and express ψ as an AR

$$\forall t_m \in \mathsf{R_m}\ (t_m[\mathsf{team}] = t_e[\mathsf{team}] \rightarrow t_e[\mathsf{arena}] = t_m[\mathsf{arena}]),$$

which asserts that if the team of the target tuple t_e is Chicago Bulls, then its arena must be United Center. As we only need to assure the consistency of the target tuple t_e, general CFDs defined on two tuples are not needed here.

6.2.2 Inferring Relative Accuracy

We next present an inference system for relative accuracy, in terms of a chase-like procedure with ARs. The chase process gives an operational semantics for ARs.

We start with some notations. Consider an entity instance I_e, a master relation I_m, and a set Σ of ARs defined on I_e and I_m. (1) We use D to denote $(I_e, \preceq_{A_1}, \ldots, \preceq_{A_n})$, *i.e.,* I_e equipped with partial orders $\preceq_{A_i}$; we use $\preceq^D_{A_i}$ to denote the partial order $\preceq_{A_i}$ in D [BKL11]. (2) We call $(D_0, t^{D_0}_e)$ the *initial instance* of e, where $\preceq^{D_0}_{A_i}$ is empty, and $t^{D_0}_e$ is the target template with $t^{D_0}_e[A_i] = \mathsf{null}$ for all $i \in [1, n]$. (3) We refer to S = (D_0,

Σ, I_m, $t_e^{D_0}$) as a *specification of entity e*. (4) We call (D, t_e^D) an *accuracy instance* of S, where t_e^D is *the target tuple template associated with* D, which is instantiated in the chase process, and may have null in some of its attributes.

In a nutshell, the chase starts with the initial instance $(D_0, t_e^{D_0})$. It deduces relative accuracy by *populating partial orders* and *instantiating the target tuple template*, yielding a sequence $(D_0, t_e^{D_0})$, $(D_1, t_e^{D_1})$, ..., $(D_m, t_e^{D_m})$ of accuracy instances. Each chase step applies an AR $\varphi \in \Sigma$ and I_m to an accuracy instance $(D_j, t_e^{D_j})$, and generates another instance $(D_{j+1}, t_e^{D_{j+1}})$. In other words, $(D_{j+1}, t_e^{D_{j+1}})$ is an updated version of $(D_j, t_e^{D_j})$, such that for some attribute A_i, either the partial order $\preceq_{A_i}^{D_{j+1}}$ extends $\preceq_{A_i}^{D_j}$ with a new pair, or $t_e^{D_j}[A_i] = \text{null}$ is instantiated by letting $t_e^{D_j+1}[A_i]$ take a value from a master tuple in I_m or a value that is already determined most accurate for A_i. The process proceeds until no changes can be made to partial orders or the target tuple template. More specifically, these are stated as follows.

(1) A single chase step. We say that $(D_{j+1}, t_e^{D_{j+1}})$ is *an immediate result* of enforcing an AR $\varphi \in \Sigma$ on $(D_j, t_e^{D_j})$ with I_m, denoted by $(D_j, t_e^{D_j}) \mapsto_\varphi (D_{j+1}, t_e^{D_{j+1}})$, if $(D_j, t_e^{D_j}) \neq (D_{j+1}, t_e^{D_{j+1}})$ and one of the following conditions holds:

(a) When $\varphi = \forall t_1, t_2 (R(t_1) \wedge R(t_2) \wedge \omega \rightarrow t_1 \preceq_{A_i} t_2)$. Then there exist tuples $t_1, t_2 \in I_e$ such that

- $(t_1, t_2) \models \omega$;
- $D_{j+1} = (I_e, \preceq_{A_1}^{D_j}, \cdots, \preceq_{A_i}^{D_j} \cup \{(t_1, t_2)\}, \cdots, \preceq_{A_n}^{D_j})$; and
- $t_e^{D_{j+1}} = (t_e^{D_j}[A_1], \cdots, \lambda(t_e^{D_j}[A_i], \preceq_{A_i}^{D_{j+1}}) \cdots, t_e^{D_j}[A_n])$.

Here $\lambda(t_e^{D_j}[A_i], \preceq_{A_i}^{D_j}) = t[A_i]$ if there exists $t \in I_e$ such that for all $t' \in I_e$, $t' \preceq_{A_i}^{D_{j+1}} t$; and it is $t_e^{D_j}[A_i]$ otherwise.

Intuitively, $\preceq_{A_i}^{D_{j+1}}$ extends $\preceq_{A_i}^{D_j}$ by including $t_1 \preceq_{A_i} t_2$, and $t_e^{D_{j+1}}[A_i]$ takes the A_i value with the highest accuracy *w.r.t.* $\preceq_{A_i}^{D_{j+1}}$ if it exists. Note that D_{j+1} and D_j agree on every attribute and partial order other than $t_e^{D_j}[A_i]$ and $\preceq_{A_i}^{D_j}$.

(b) When φ = $\forall t_m\ (R_m(t_m) \wedge \omega \rightarrow t_e[A_i] = t_m[A_i])$. Then there exist tuples $t \in I_e$ and $t_m \in I_m$ such that

- $(t_e, t_m) \models \omega$, $D_{j+1} = D_j$, and
- $t_e^{D_{j+1}}$ = $(t_e^{D_j}[A_1], \cdots, t_m[A_i], \cdots, t_e^{D_j}[A_n])$.

Here $t_e^{D_{j+1}}$ differs from $t_e^{D_j}[A_i]$ only in attribute A_i by taking master data $t_m[A_i]$, while D_{j+1} remains unchanged from D_j.

We say that $(D_j, t_e^{D_j}) \mapsto_\varphi (D_{j+1}, t_e^{D_{j+1}})$ is *valid* if (a) there exist no t_1 and t_2 such that both $t_1 \prec_{A_i}^{D_{j+1}} t_2$ and $t_2 \prec_{A_i}^{D_{j+1}} t_1$ (*i.e.*, $t_1 \preceq_{A_i}^{D_{j+1}} t_2$, $t_2 \preceq_{A_i}^{D_{j+1}} t_1$ but $t_1[A_i] \neq t_2[A_i]$), and

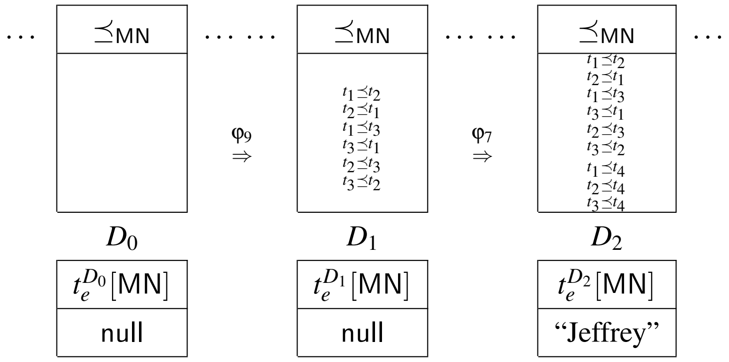


Figure 6.2: Single chase steps

(b) $t_e^{D_j}[A_i]$ is not changed if $t_e^{D_j}[A_i] \neq \mathsf{null}$. In the chase process we consider valid chase steps only.

Observe the following: (a) the entity instance I_e and the master data I_m remain unchanged when ARs are enforced; (b) $\preceq_{A_i}^{D_j}$ and $\preceq_{A_i}^{D_{i+1}}$ are partial orders for all attributes A_i, such that for all $t_1, t_2 \in I_e$, if $t_1 \preceq_{A_i}^{D_j} t_2$ and $t_2 \preceq_{A_i}^{D_j} t_1$ then $t_1[A_i] = t_2[A_i]$; and (c) if $t_e^{D_j}[A_i] \neq \mathsf{null}$, then $t_e^{D_{j+1}}[A_i] = t_e^{D_j}[A_i]$, *i.e.,* all non-null values of $t_e^{D_j}$ *remain unchanged.*

Example 6.2.2: Consider I_e = stat (Table 6.1), I_m = nba (Table 6.2), and Σ consisting of the ARs given in Example 6.2.1. Let D_0 be I_e with empty partial orders, and $t_e^{D_0}$ be the initial target template with $t_e^{D_0}[A]$ = null for all attributes A. After enforcing φ_9 on $(D_0, t_e^{D_0})$, it yields $(D_1, t_e^{D_1})$ as the first step in Fig. 6.2, in which $\preceq_{\mathsf{MN}}$ is extended on (t_1, t_2), (t_2, t_3) and (t_3, t_1). Similarly, after enforcing φ_7 on $(D_1, t_e^{D_1})$, it yields $(D_2, t_e^{D_2})$ as the second step in Fig. 6.2, which extends $\preceq_{\mathsf{MN}}$ on (t_1, t_4) and instantiates $t_e^{D_2}[\mathsf{MN}]$ = "Jeffrey". □

(2) Chase. A *chasing sequence* of D_0 by Σ and I_m is a sequence of accuracy instances $(D_0, t_e^{D_0})$, $(D_1, t_e^{D_1})$, ..., $(D_l, t_e^{D_l})$, ..., where for each $i \geq 1$, there exists some AR $\varphi \in \Sigma$ such that $(D_j, t_e^{D_j}) \mapsto_\varphi (D_{j+1}, t_e^{D_{j+1}})$ is valid.

A chasing sequence $(D_0, t_e^{D_0})$, ..., $(D_k, t_e^{D_k})$ is said to be *terminal* if it is finite and moreover, no more valid step can be enforced on $(D_k, t_e^{D_k})$. We refer to $t_e^{D_k}$ as a *deduced target tuple* of specification S, and $(D_k, t_e^{D_k})$ as the *terminal instance* of the chasing sequence.

Intuitively, the chase repeatedly applies ARs to deduce relative accuracy and instantiate the target tuple template, until it reaches an instance that cannot be further changed.

Example 6.2.3: Consider $D_0, t_e^{D_0}, \Sigma$ and I_m given in Example 6.2.2. By enforcing ARs

φ_9, φ_7, φ_5, φ_{10}, φ_6, φ_1, φ_4, φ_2, φ_3 and φ_{11} on $(D_0, t_e^{D_0})$ in this order with I_m, one can get a deduced target tuple t_e[FN, MN, LN, rnds, totalPts, J#, league, team, arena] = (Michael, Jeffrey, Jordan, 27, 772, 23, NBA, Chicago Bulls, United Center). Note that t_e is a complete target tuple from stat, which draws values from *different tuples*, *e.g.,* t_2 and t_4 of stat and s_1 of nba. □

6.3 Fundamental Problems

Given a specification $S = (D_0, \Sigma, I_m, t_e^{D_0})$ of an entity e, we want to know whether chasing on D_0 by Σ and I_m terminates? Whether will all chasing sequences of D_0 lead to the same deduced target tuple t_e? When the target tuple t_e is incomplete, can we make it complete while observing the ARs in Σ? Can we find top-k candidate targets for users to chose? This section studies these issues. As will be seen in Section 6.4, our framework for deducing relative accuracy and target tuples are based on these results.

(1) Termination of chase. Is every chasing sequence of D_0 by Σ and I_m an initial subsequence of a terminal chasing sequence? The answer to this question is affirmative.

Proposition 6.3.1: *Every chasing sequence of D_0 by Σ and I_m is finite and leads to a terminal instance in $O(|I_e|^2)$ steps, where $|I_e|$ is the size of the entity instance I_e in D_0.* □

Proof sketch: Each chase step expands a partial order or instantiates an attribute $t_e[A_i]$ that had a null value. Moreover, $\preceq_{A_i}$ remains to be a partial order and $t_e[A_i]$ does not change from one non-null value to another. Hence chasing always terminates. Further, the size of all the partial orders is bounded by $|I_e|^2$, and the arity of t_e is no larger than $|I_e|$. From these the bound on chase steps follows. □

(2) The Church-Rosser property. Another question asks whether different terminal chasing sequences of D_0 by Σ and I_m lead to *the same unique terminal instance*, no matter what rules in Σ are used and in what order they are applied. This is known as the *Church-Rosser property* (see, *e.g.,* [AHV95]). If a specification S has the Church-Rosser property, we say that *S is Church-Rosser*. Obviously if S is Church-Rosser, then the uniquely deduced target tuple is *deterministic*, yielding a unique target that can be "trusted".

Unfortunately, not all specifications are Church-Rosser.

Example 6.3.1: Consider the specification S described in Example 6.2.3. One can verify that S is Church-Rosser. However, let us extend S to S' by adding an extra rule φ_{12}: $\forall t_1, t_2 \in \mathsf{stat}\ \big(t_1[\mathsf{league}] = \text{NBA} \wedge t_2[\mathsf{league}] = \text{SL} \rightarrow t_1 \preceq_{\mathsf{league}} t_2\big)$. Then S' is not Church-Rosser. Indeed, there are two chasing sequences that deduce different target tuples: one is the sequence given in Example 6.2.3 with $t_e[\mathsf{league}] = \text{NBA}$, and the other is by enforcing ARs φ_7, φ_5, φ_{10} and φ_{12} in this order, yielding a target tuple t'_e with $t'_e[\mathsf{league}] = \text{SL}$. □

This tells us that if S is not Church-Rosser, it may lead to multiple conflicting targets (*e.g.,* t_e and t'_e on league), which cannot be accurate at the same time. Thus specifications that are not Church-Rosser should be *identified and revised.*

To do this, we provide *a necessary and sufficient condition* for deciding whether a specification S is Church-Rosser. We say that a terminal chasing sequence $(D_0, t_e^{D_0})$, $\ldots$, $(D_k, t_e^{D_k})$ is *stable* if for all *invalid* chase steps that enforce an AR φ on $(D_k, t_e^{D_k})$, φ cannot be enforced on $(D_j, t_e^{D_j})$ as a valid step for all $j \in [0, k-1]$. That is, suppose that $(D_k, t_e^{D_k})$ can be further changed by φ by letting $t_e^{D_k}[A_i]$ change from a non-null value to another, or by allowing both $t_1 \preceq_{A_i}^{D_k} t_2$ and $t_2 \preceq_{A_i}^{D_k} t_1$ while $t_1[A_i] \neq t_2[A_i]$. Then the change cannot be inflicted to any $(D_j, t_e^{D_j})$ as a valid move. Intuitively, if φ could be enforced as a valid step, it would lead to a terminal sequence different from $(D_k, t_e^{D_k})$.

A stable chasing sequence prevents any conflicts in the chase such as those in Example 6.3.1, and allows us to efficiently determine whether S is Church-Rosser. In light of this, in the sequel we focus on Church-Rosser specifications only.

Theorem 6.3.2: *Given a specification $S = (D_0, \Sigma, I_m, t_e^{D_0})$, (a) S is Church-Rosser if and only if there exists a terminal chasing sequence of S that is stable; and (b) it is in $O((|I_e|^2 + |I_m|)|\Sigma|)$ time to decide whether S is Church-Rosser.* □

Proof sketch: We prove (1) here, and defer the proof of (2) to Section 6.5, where we will provide a checking algorithm.

First assume that S is Church-Rosser. Suppose by contradiction that S has no stable terminal sequence. Then there exist a chasing sequence $\overline{Q_1}$ and an AR φ such that φ is an invalid chase step for the terminal instance of $\overline{Q_1}$ but is valid for an intermediate instance of $\overline{Q_1}$. Then one can construct another sequence $\overline{Q_2}$ leading to a different terminal instance, by applying φ to the intermediate instance of $\overline{Q_1}$. This contradicts the assumption that S is Church-Rosser.

Conversely, assume that S is not Church-Rosser but there exists a stable chasing sequence $\overline{Q_1}$, leading to a terminal instance $(D_k, t_e^{D_k})$. Since S is not Church-Rosser, there must exist another chasing sequence $\overline{Q_2}$ to a different terminal instance. Let

$(D'_l, t_e^{D'_l})$ be the first instance in $\overline{Q_2}$ that contradicts $(D_k, t_e^{D_k})$, by applying an AR φ, which either adds $t_1 \preceq_A t_2$ to D'_l that is not in D_k, or by instantiating $t_e^{D'_l}[A]$ with a value $v \neq t_e^{D_k}[A]$. Then one can verify that φ is an invalid chase step for $(D_k, t_e^{D_k})$ but is valid for some instance in $\overline{Q_1}$. Hence $\overline{Q_1}$ is not stable, a contradiction. □

(3) Deducing candidate targets. When S is Church-Rosser, its deduced target tuple t_e may still be *incomplete*, *i.e.,* some attributes remain null. For example, if we drop AR φ_{11} of Example 6.2.1 from the specification of Example 6.2.3, the reduced specification is still Church-Rosser, but its deduced target is incomplete since the most accurate value of arena can no longer be determined, as indicated in Example 6.1.2.

This gives rise to the following question: can we find candidate targets and suggest them for the users to consider? More specifically, a complete tuple t'_e is called a *candidate target* of a specification S = (D_0, Σ, I_m, $t_e^{D_0}$) if

- for each attribute A_i, $t'_e[A_i] = t_e[A_i]$ if $t_e[A_i] \neq \text{null}$, and $t'_e[A_i]$ is a value in $\text{dom}(A_i)$ otherwise, where t_e is the unique deduced target tuple of S;
- $S' = (D_0, \Sigma, I_m, t'_e)$ is Church-Rosser and moreover, t'_e is the deduced target tuple of S'.

That is, a candidate target t'_e keeps the non-null values of t_e unchanged but instantiates those null attributes of t_e. Moreover, when we treat t'_e as the initial target template, the chase verifies that t'_e "satisfies" the constraints imposed by the ARs of Σ, and is deduced as the target tuple of S'.

The *candidate target problem* is to determine, given a specification S of an entity that is Church-Rosser, whether there exists a candidate target t'_e of S. It is, however, nontrivial.

Theorem 6.3.3: *The candidate target problem is NP-complete. It remains NP-hard for specifications S = (D_0, Σ, I_m, $t_e^{D_0}$) in which Σ consists of ARs of form (1) only, and when candidate targets t_e of S take values from I_e and I_m only.* □

Proof sketch: One can verify that the problem in NP-hard by reduction from the 3SAT problem, which is NP-complete (cf. [Pap94]). The reduction uses AR of form (1) and constructs candidate targets using values from I_e and I_m only. To show the upper bound, we first establish a small model property: if there exists a candidate target of S, then there exists one composed of values from a set V, where V consists of values from I_e, I_m and a bounded number of constants. We then give an NP algorithm that first guesses a tuple t'_e with values from V, and then checks whether t'_e is the deduced target tuple from (D_0, t'_e) in PTIME by Theorem 6.3.2. □

The number of candidates t'_e for a Church-Rosser S could be quite large, exponential or even infinite.

Example 6.3.2: Consider $R = (A_1, \ldots, A_n)$, an entity instance I_e of R with tuples $t_1 = (0, \ldots, 0)$ and $t_2 = (1, \ldots, 1)$, and empty Σ and I_m. Then there are 2^n candidate targets with values from $\{0, 1\}$, *i.e.,* each tuple $t \in \{0, 1\}^n$ is a candidate target. Worse still, if some A_i of R has an infinite domain, there are possibly *infinitely many* candidate targets. □

(4) Finding top-*k* candidate targets. It is infeasible to enumerate all candidate targets. This suggests that we find top-k candidate targets for S based on a preference model.

We specify the preference model as a pair $(k, p(\cdot))$, where k is a natural number, and $p(\cdot)$ is a monotone scoring function such that given a set T_e of candidate targets, $p(T_e)$ is a real number. To simplify the discussion we assume that a real number $w_{A_i}(v)$ is associated with each value v in domain $\mathsf{dom}\ (A_i)$ (if $\mathsf{dom}\ (A_i)$ is infinite, $w_{A_i}(v)$ is the same for all v outside of I_e and I_m), referred to as the *score of* v. The score could be placed by the users as the confidence in v [FG12], found as probabilities by truth discovery algorithms [BCMP10, GAMS10, WM11, YHY08, ZRGH12] (see Section 6.7), or automatically derived by counting the occurrences of v in the A_i column and from co-existence of attributes and available scores. We define the *score*

$$p(T_e) = \sum_{t'_e \in T_e} \sum_{A_i \in \mathsf{attr}(R)} w_{A_i}(t'_e[A_i]).$$

Such preference is often too "soft" to be modeled as ARs or partial orders, and candidates derived from it may not be as "deterministic" ("certain") as deduced targets by the chase. Nonetheless, users often find such candidates helpful, as commonly practiced in data repairing heuristics [FG12].

For a Church-Rosser S, a preference model (k, $p(\cdot)$) and a number C, the *top-k candidate problem* is to decide whether there exists a set T_e of k candidate targets with $p(T_e) \geq C$.

Theorem 6.3.4: *The top-k candidate problem is NP-complete, and NP-hard under the same restriction of Theorem 6.3.3.* □

Proof sketch: The lower bound is verified by reduction from the MAX 3SAT problem (the decision version), which is NP-complete (cf. [Pap94]). The reduction uses ARs of form (1), and candidate targets with values from I_e and I_m only. We show that the problem is in NP based on a small model property similar to the one given in the proof of Theorem 6.3.3. □

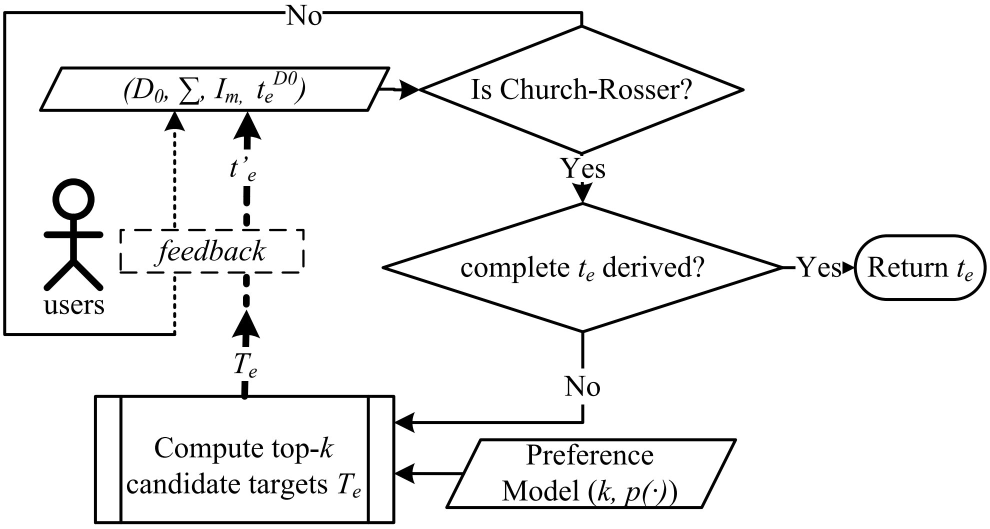

No
$(D_0, \sum, I_m, t_e^{D0})$
Is Church-Rosser?
Yes
t'_e
feedback
users
complete t_e derived?
Yes
Return t_e
No
T_e
Compute top-k candidate targets T_e
Preference Model $(k, p(\cdot))$

Figure 6.3: Framework overview

6.4 A Framework for Deducing Target Tuples

We now present a framework for deducing complete target tuples for entities. As depicted in Fig. 6.3, given a specification $S = (D_0, \Sigma, I_m, t_e^{D_0})$ of an entity e, it populates partial orders for relative accuracy and instantiates the target tuple template, based on the chase given in Section 6.2. It automatically deduces as many accurate values for e as possible, and interacts with the users to revise candidate targets, until a complete target tuple is found. It works as follows.

(1) *Church-Rosser checking.* It first inspects whether S is Church-Rosser via *automated reasoning*. The Church-Rosser property warrants a unique target tuple, in which the accurate values can be trusted (Section 6.3). If S is not Church-Rosser, the users are invited to revise S (see step (4) below), by following the "No" branch. The revised S is then checked.

(2) *Computing target tuple t_e.* When S is confirmed Church-Rosser, the framework computes the unique deduced target tuple t_e by means of the chase. It returns t_e if it is complete. Otherwise it computes a top-k set of candidate targets.

(3) *Computing top-k candidate targets.* As remarked in Section 6.3, t_e may be incomplete, and it is hard to identify (all) candidate targets (Theorem 6.3.3 and Example 6.3.2). To this end, the framework computes a top-k set T_e of candidate targets. It comes up with a preference model $(k, p(\cdot))$ following the practice of data repairing heuristics, which the users may opt to adjust. Based on $(k, p(\cdot))$, it computes T_e with k

tuples such that (a) for each $t'_e \in T_e$, t_e is a candidate target of S, and (b) for all sets T'_e with k candidate targets, $p(T_e) \geq p(T'_e)$, *i.e.,* tuples in T_e have the highest scores. When there exist at least k candidate targets of S, T_e consists of k distinct tuples; otherwise T_e includes all candidate targets of S. The set T_e is then suggested to the users.

(4) *User feedback.* The users are invited to inspect T_e. They may opt to choose some $t'_e \in T_e$ as the target tuple (recall that for each candidate target t'_e, $t'_e[A] = t_e[A]$ if $t_e[A] \neq \mathsf{null}$); or revise S by instantiating $t_e[B]$ with either the value of some $t'_e[B]$ or a value $v \in \mathsf{dom}(B)$, for some $t_e[B] = \mathsf{null}$. The users are also allowed to revise S by editing ARs in Σ and tuples in I_e (D_0). The revised S with the designated initial values is then checked by step (1).

The process proceeds until a complete t_e is found.

In the rest of the chapter we will provide algorithms underlying the framework: an algorithm for checking the Church-Rosser property of S and deducing t_e in Section 6.5, and algorithms for computing top-k candidate targets in Section 6.6.

Remark. (1) To find ARs as input of the framework, we need algorithms for discovering ARs from (*possibly dirty*) data. ARs of type (2) can be discovered along the same lines as matching dependencies (see, *e.g.,* [FG12] for a survey). ARs of type (1) could be found by mining first-order logic rules (*e.g.,* [FL01]). Given a relation r of schema R, one may also group pairs of its tuples (t_i, t_j) into classes based on their attribute values $(t_i[A], t_j[A])$ $(t_i, t_j \in r,\ A \in \mathsf{attr}(R))$ to denote accuracy orders, and discover ARs by analyzing the containment of those classes via a level-wise approach (*e.g.,* [CM08]). We defer a full treatment of AR discovery to future work.

(2) The framework can handle *possibly dirty* entity instances. Indeed, constant CFDs [BFG$^+$07] for detecting data inconsistencies can be expressed as ARs (Section 6.2). Thus in the same framework the consistency of target tuples can be assured. The framework can also incorporate data repairing algorithms, which have been well studied (see, *e.g.,* [FG12]).

6.5 Checking the Church-Rosser Property

We next present an algorithm that, given a specification $S = (D_0, \Sigma, I_m, t_e^{D_0})$, checks whether S is Church-Rosser. If so, it computes the unique terminal instance $(D,\ t_e)$, and returns *nil* otherwise. The algorithm is in $O((|I_e|^2 + |I_m|)|\Sigma|)$ time, and thus gives a constructive proof for Theorem 6.3.2(2).

The algorithm is denoted by IsCR and shown in Fig. 6.4. Following Theorem 6.3.2(1), IsCR checks whether S has a stable terminal chasing sequence, by simulating the chase. At each step of the chase, it collects all valid steps in a set $\mathcal{Q}$, and when the chase process proceeds, it checks whether any valid step in $\mathcal{Q}$ becomes invalid. If so, it concludes that S is not Church-Rosser. Indeed, an invalid step for an instance *remains invalid* in the rest of the chasing sequence. Hence if a valid step becomes invalid later, it will not lead to any stable terminal chasing sequence. If IsCR inspects all valid steps and if none of them becomes invalid, it actually identifies a stable terminal sequence, and thus concludes that S is Church-Rosser. In the process (D, t_e) is also constructed.

Algorithm IsCR makes use of (1) a procedure, denoted by Instantiation, as a preprocessing step to identify all single chase steps, and (2) an indexing structure $\mathcal{H}$ for efficiently locating applicable chase steps, described as follows.

Computing single chase steps. Procedure Instantiation pre-computes possible single chase steps, collected in a set Γ, by partially evaluating each AR $\varphi \in \Sigma$ on tuples in the entity instance I_e of D_0 and master relation I_m, as follows.

(1) When φ is of form (1) $\forall t_1,t_2(\omega \rightarrow t_1 \preceq_{A_k} t_2)$, for each pair (t_i,t_j) of tuples in I_e, it computes $\phi = (\omega' \rightarrow t_i \preceq_{A_k} t_j)$, where ω' is obtained by evaluating $\omega(t_i,t_j)$, which substitutes (t_i,t_j) for (t_1,t_2) in ω. More specifically, for each predicate in ω, (a) if it is of the form $t_1[A_l]$ op $t_2[A_l]$ or $t_s[A_s]$ op c $(s \in [1,2])$, where op is one of $=, \neq, >, <, \leq, \geq$, then the predicate on (t_i,t_j) evaluates to true or false. If it is true, the predicate is not included in ω'. If it is false, ω' also becomes false. (b) If it is $t_1 \prec_{A_l} t_2$ (resp. $t_1 \preceq_{A_l} t_2$), the predicate is replaced by $t_i \prec_{A_l} t_j$ (resp. $t_i \preceq_{A_l} t_j$) in ω'. We include ϕ in Γ if ω' is not false. Intuitively, ϕ indicates a single chase step: if ω' is satisfied, then $t_i \preceq_{A_k} t_j$ could be deduced.

(2) When $\varphi = \forall t_m\ (R_m(t_m) \wedge \omega \rightarrow t_e[A_i] = t_m[B])$, *i.e.,* of form (2), for each $t' \in I_m$, it computes $\phi = (\omega' \rightarrow t_e[A_i] = c)$, where t_e is the target template, c is the constant $t'[B]$, and ω' is obtained from $\omega(t_m)$ by substituting constant $t'[B']$ for each $t_m[B']$ in ω. We include ϕ in Γ, which indicates that if ω' is satisfied, then $t_e[A_i]$ can be instantiated with c.

Note that no ϕ in Γ carries the $\forall$ quantifier. Moreover, each chase step can be carried out by enforcing some ϕ in Γ rather than ARs in Σ. We use LHS(ϕ) to denote ω'.

Example 6.5.1: The following single chase steps can be derived: (a) *true*→ $16 \prec_{\mathsf{rnds}} 27$

from t_1, t_2 of Table 6.1 and φ_1 (form (1)) of Table 6.3; (b) $16 \prec_{\mathsf{rnds}} 27 \rightarrow 45 \prec_{\mathsf{J\#}} 23$ from t_1, t_2 and φ_2 (form (1)); and (c) $t_e[\mathsf{FN}, \mathsf{LN}]$ = (Michael, Jordan) $\rightarrow$ $t_e[\mathsf{league}, \mathsf{team}]$ = (NBA, Chicago Bulls) from master tuple s_1 of Table 6.2 and φ_6 (form (2)) of Table 6.3. □

Building Indices. Algorithm IsCR uses an indexing structure $\mathcal{H}$ to speed up the process of finding next applicable chase step. The structure $\mathcal{H}$ is defined as follows.

(1) For each $\phi \in \Gamma$, $\mathcal{H}$ contains a counter n_ϕ to keep track of the number of predicates in $\mathsf{LHS}(\phi)$ that are not yet satisfied.

(2) For each predicate δ of the form of either $t_i \preceq_{A_k} t_j$ or $t_e[A_k] = c$, $\mathcal{H}$ maintains a set $\Phi_\delta = \{\phi \mid \phi \in \Gamma \wedge \delta \in \mathsf{LHS}(\phi)\}$, *i.e.,* the set of ϕ's in Γ that contain δ in $\mathsf{LHS}(\phi)$.

(3) A set $\mathcal{Q}$ is maintained by $\mathcal{H}$, which consists of all applicable single chase steps that were once valid. Initially, $\mathcal{Q} = \{\phi \mid \phi \in \Gamma \wedge n_\phi = 0 \wedge \phi$ was a valid step for $(D_0, t_e^{D_0})\}$.

Algorithm. We new present the main driver of IsCR. Given S, it first identifies all possible single steps and builds the index $\mathcal{H}$, by invoking procedures Instantiation and InitIndex (not shown), respectively (lines 1-2). It then initializes the accuracy instance template (D, t_e) with $(D_0, t_e^{D_0})$ (line 3).

After these, IsCR simulates the chasing of S (lines 3-13). When the set $\mathcal{Q}$ in $\mathcal{H}$ of valid steps is nonempty, it picks an applicable step ϕ from $\mathcal{Q}$ using a procedure NextStep (line 5, not shown), which removes ϕ from $\mathcal{Q}$. It then enforces ϕ as follows. If ϕ is derived from an AR of form (1) (lines 6-8), IsCR adds the derived partial order to D (line 7), and deduces $t_e[A_k]$ whenever possible (line 8). If ϕ is derived from an AR of form (2) (lines 9-10), IsCR sets $t_e[A_k] := c$ (line 10). If ϕ is invalid for (D, t_e), we can conclude that S is not Church-Rosser since there will be no stable terminal chasing sequences, as argued above, and IsCR returns *nil* (line 11). Otherwise IsCR updates $\mathcal{H}$ to reflect the changes to D and t_e (line 12): for each $t_i \preceq_{A_k} t_j$ derived, $n_{\phi'}$ is decreased by 1 for each $\phi' \in \Phi_{t_i \preceq_{A_k} t_j}$; and for each $t_e[A_k] = c$ derived, it decreases $n_{\phi'}$ by 1 for each $\phi' \in \Phi_{t_e[A_k]=c}$. For any ϕ' with $n_{\phi'} = 0$, ϕ' is added to $\mathcal{Q}$, *i.e.,* it now becomes a valid chase step, to be considered later. The process proceeds until no more steps in $\mathcal{Q}$ need to checked (line 13), and it returns (D, t_e) as the terminal instance (line 14).

Correctness & complexity. The correctness of IsCR follows from Theorem 6.3.2(1) and the argument above, since it checks all possible chase steps that are valid at some point of chasing. For the complexity, observe the following. (1) Instantiation is in $O(|\Sigma|(|I_e|^2 + |I_m|))$ time, which is also the bound on $|\Gamma|$. After Instantiation, IsCR no

```
Input: A specification $S = (D_0, \Sigma, I_m, t_e^{D_0})$.
Output: The unique terminal instance $(D, t_e)$ if $S$ is Church-Rosser,
        and nil otherwise.

1.  $\Gamma := \mathsf{Instantiation}(D_0, \Sigma, I_m)$;
2.  $\mathcal{H} := \mathsf{InitIndex}(\Gamma, D_0, t_e^{D_0})$; /*$\mathcal{Q}$ in $\mathcal{H}$ maintains single steps*/
3.  $D := D_0$; $t_e := t_e^{D_0}$;
4.  repeat
5.    $\phi := \mathsf{NextStep}(\mathcal{H})$;
6.    if $\phi = (\omega \to t_i \preceq_{A_k} t_j)$ (i.e., form (1)) then
7.      $D$ := the transitive closure of $D \cup \{t_i \preceq_{A_k} t_j\}$;
8.      Update $t_e[A_k]$;
9.    if $\phi = (\omega \to t_e[A_i] = c)$ (i.e., form (2)) then
10.     $t_e[A_i] := c$;
11.   if not $\mathsf{IsValid}(\phi, D, t_e)$ then return nil;
12.   Update $\mathcal{H}$;
13. until $\mathcal{Q}$ in $\mathcal{H}$ becomes empty $\emptyset$;
14. return $(D, t_e)$
```

Figure 6.4: Algorithm IsCR

longer needs to visit I_e. (2) With the indices, NextStep takes O(1) time, and each ϕ is checked only once. (3) Each step derives new partial orders $\preceq_{A_k}$ and/or instantiates $t_e[A_k]$. Thus, the total number of steps processed (lines 5-12) is bounded by $O(|I_e|^2)$. Therefore, IsCR is in $O(|\Sigma|(|I_e|^2 + |I_m|))$ time. As remarked earlier, I_e is *much smaller* than a database instance. As will be seen in Section 6.7, IsCR takes about 10ms.

6.6 Computing Top-k Candidate Targets

We next provide algorithms that, given a Church-Rosser specification $S = (D_0, \Sigma, I_m, t_e^{D_0})$ and a preference model $(k, p(\cdot))$, compute a set T_e of top-k candidate targets. Here T_e consists of k distinct candidate targets of S with the maximum score $p(T_e)$ if there exist at least k such tuples of S; and otherwise T_e includes all candidate targets of S.

Theorem 6.3.4 tells us that the top-k candidate problem is NP-complete. Worse still, it is impossible to find a PTIME algorithm for it with a bounded approximation-ratio.

Theorem 6.6.1: *The top-k candidate targets problem (optimization version) is* NPO-*complete.* □

Here NPO is the *class* of all NP optimization problems. An NPO-complete problem is NP-hard to optimize, and is among the hardest optimization problems.

Proof sketch: The problem is in NPO since its decision version is in NP (Theorem 6.3.4). We show that it is NPO-hard by L-reduction from the MAXIMUM WEIGHTED 3SAT problem, which is known to be NPO-complete (cf. [Aus99]). □

Despite the hardness, we provide three algorithms to find top-k candidate targets, all with the *early termination* property, *i.e.,* they stop as soon as top-k candidate targets are found. The first two are exact algorithms. (1) The first one, RankJoinCT, extends prior algorithms [IAE04, SP08] for computing top-k joins of ranked lists (Section 6.6.1). (2) The second one, TopKCT, is developed for a more general setting when the ranked lists are *not* given (Section 6.6.2). We show that TopKCT incurs *lower cost* than RankJoinCT, and is *instance optimal w.r.t.* the number of visits to the data needed. (3) The third one, $\mathsf{TopKCT_h}$, is a PTIME heuristic version of TopKCT (Section 6.6.3). We also identify special cases when the top-k candidate targets problem is in PTIME.

6.6.1 RankJoinCT: An Algorithm based on Rank Join

Given a set of ranked lists and a monotone scoring function, *the top-k rank join problem* is to compute the top k join results of the lists with the highest scores. Our problem can be modeled as an extension of the top-k rank join problem as follows. Consider a Church-Rosser specification S, of which t_e is the unique deduced target tuple. Let Z be the set of attributes A such that $t_e[A_i] = \mathsf{null}$. Assume *w.l.o.g.* that Z consists of m attributes $A_1, \cdots, A_m$. Then a set T_e of top-k candidate targets is a set of top-k join results of values in each domain of Z such that it satisfies an *additional condition*: for each $t \in T_e$, the revised specification $S' = (D_0, \Sigma, I_m, t'_e)$ must also be Church-Rosser, where $t'_e[Z] = t[Z]$ and $t'_e[B] = t_e[B]$ for all $B \in R \setminus Z$.

In light of this, we develop algorithm RankJoinCT by extending top-k rank join algorithms [IAE04, SP08]. The algorithm assumes that the domain values of Z attributes are *ranked* based on their scores (see $w_{A_i}(\cdot)$ in Section 6.3). It takes as input S, (k, $p(\cdot)$),

t_e and moreover, m lists $L_1, \cdots, L_m$, such that L_i is the ranked list of values in the *active domain* of A_i for all $i \in [1,m]$. It returns a top-k *list* T_e of candidate targets as required. Note that L_i is *finite*: the active domain of A_i is $\mathsf{dom}(A_i)$ if it is finite; otherwise it includes all A_i values from I_e or I_m and at most one more distinct value from $\mathsf{dom}(A_i)$, which suffices to denote values outside of I_e or I_m and is referred to as a *default value* (see Section 6.3). We omit the details of RankJoinCT for the lack of space, but give an example below to illustrate how it works.

Example 6.6.1: Consider the specification S of Example 6.2.3 and a preference model $(k = 2, p(\cdot))$, where $p(\cdot)$ counts value occurrences. Suppose that we drop team from φ_6 of Table 6.3. Then the deduced target t_e is incomplete since $t_e[\mathsf{team}]$ and $t_e[\mathsf{arena}]$ become null. To find top 2 candidate targets for S, RankJoinCT takes as input the modified S, $(2, p(\cdot))$ and two ranked list L_{team} = [Chicago Bulls; Chicago; Birmingham Barons; $\perp_{\mathsf{team}}$] and L_{arena} = [United Center; Chicago Stadium; Regions Park; $\perp_{\mathsf{arena}}$], where $\perp_{\mathsf{team}}$ and $\perp_{\mathsf{arena}}$ are default values. It maintains upper bounds u_{team} and u_{arena}, asserting that for all tuples with values in L_{team} (resp. L_{arena}), their scores are no higher than u_{team} (resp. u_{arena}).

RankJoinCT iteratively retrieves candidate targets from the two lists. Initially, it picks Chicago Bulls for team and United Center for arena, and forms a candidate target t_1 with $t_1[\mathsf{team}, \mathsf{arena}]$ = (Chicago Bulls, United Center) ($p(t_1)$ = 4). It then updates both u_{team} and u_{area} to 1 + 2 = 3. After that, it picks the next top unseen value v from L_{arena}, *i.e.,* either Chicago Stadium or Regions Park, and chooses a candidate target from the join results of $\{v\}$ with other fixed values in t_1, which is t_2 with $t_2[\mathsf{team}, \mathsf{arena}]$ = (Chicago Bulls, Chicago Stadium (or Regions Park)) ($p(t_2)$ = 2 + 1 = 3). Now $p(t_1)$ and $p(t_2)$ are no less than u_{FN} and u_{arena}.

Further, for each $t \in \{t_1, t_2\}$, RankJoinCT has to check the condition additional to [IAE04, SP08], *i.e.,* whether t is a candidate target. This is done by procedure check, which is essentially IsCR of Fig. 6.4 by taking t as the initial target. As t_1 and t_2 pass check, they are returned as top-2 candidates. □

Following [IAE04], one can readily verify:

Proposition 6.6.2: RankJoinCT *finds top-k candidate targets with the early termination property,* i.e., *it does not need to check all tuples in the product of ranked lists.* □

However, RankJoinCT is not ideal. (1) In practice, domain values are often *not* given in ranked lists, and sorting the domains is costly. (2) RankJoinCT invokes pro-

```
Input: $S$, $(k, p(\cdot))$ and $t_e$ as for RankJoinCT, and heaps $H_1, \ldots, H_m$.
Output: A list $T_e$ of top-$k$ candidate targets of $S$.

$T_e := nil$;
for each $i \in [1,m]$ do $B_i := [H_i$.pop()$]$;
$o.t[Z] := (B_1[0], \cdots, B_m[0])$; $o.t[R \setminus Z] := t_e[R \setminus Z]$;
for each $i \in [1,m]$ do $o.p_i := 0$ ;
$o.w := p(\{o.t\})$;
$T := \{o.t\}$; $Q$ := BrodalQueue$(\{o\})$;
while $\|T_e\| < k$ and $\|Q\| > 0$ do
    $o := Q$.pop();
    if check$(o.t, S)$ then $T_e$.append$(o.t)$; /* add $o.t$ to $T_e$*/
    for each $i \in [1,m]$ do
        if len$(B_i) \leq o.p_i + 1$ then $B_i$.append$(H_i$.pop()$)$;
        $o' := o$;
        $o'.p_i := o'.p_i + 1$; $o'.t[A_i] := B_i[o'.p_i]$;
        $o'.w := o.w - w_{A_i}(o.t[A_i]) + w_{A_i}(o'.t[A_i])$;
        if $o'.t \notin T$ then $Q$.push$(o')$; $T := T \cup \{o'.t\}$;
return $T_e$;
```

Figure 6.5: Algorithm TopKCT

cedure check for each tuple in the join result (Example 6.6.1); this yields *exponentially* many calls, each taking $O(|\Sigma|(|I_e|^2 + |I_m|))$ time.

6.6.2 TopKCT: A Brodal Queue Based Algorithm

To remedy the problems of RankJoinCT, we next present TopKCT. In contrast to RankJoinCT, TopKCT does not require ranked lists as input, and invokes check much less.

TopKCT maintains several structures: (1) a heap H_i for each $A_i \in Z$, to store the values in the active domain of A_i; it is able to pop up the top value in H_i in $O(\log |H_i|)$ time, and can be pre-constructed in linear time; (2) a Brodal queue Q, to keep track of tuples to be checked; Q is a worst-case efficient priority queue [Bro96]; it takes $O(1)$

time to insert a tuple and $O(\log |Q|)$ time to pop up the top tuple; and (3) a hash set T to record tuples that were once pushed to Q.

TopKCT is shown in Fig. 6.5. Its input includes S, (k, $p(\cdot)$), t_e as for RankJoinCT; but instead of ranked lists, it takes m heaps $H_1, \cdots, H_m$ as input. It computes a top-k list T_e like RankJoinCT. The key idea behind TopKCT is that when T_e is nonempty, if t is the next best tuple, then there must exist a tuple $t' \in T_e$ such that t and t' differ in *only one* attribute. Hence it capitalizes on the heaps to pop up a tuple that is guaranteed to be the next best, one at time, rather than to compute costly ranked joins. The tuple is then validated by check, and is added to T_e if it is a candidate target. The process proceeds until either T_e is found or the search space is exhausted, with *the early termination property*.

More specifically, TopKCT first lets T_e be empty (line 1). It then pops the top values out of the m heaps $H_1, \cdots, H_m$ to m vectors $B_1, \cdots, B_m$, respectively (line 2), where B_i is a buffer of the values from H_i for $i \in [1,m]$. Note that $t_Z = (B_1[0], \cdots, B_m[0])$ is the tuple with the highest score. An object o is then formed (lines 3-5), with $2+m$ members: (1) $o.t$ is a tuple t_0, where $t_0[Z] = t_Z$ and $t_0[R \setminus Z] = t_e[R \setminus Z]$; (2) for $i \in [1,m]$, $o.p_i$ is an integer, initially 0; it is an index of B_i, indicating that $o.t[A_i]$ takes value from $B_i[o.p_i]$; and (3) $o.w$ is a real number, which is the score of $o.t$. The Brodal queue Q is initialized with $\{o\}$ only, and it lets the hash set $T := \{o.t\}$ (line 6). Note that when Q has multiple objects, it always pops up o' with the highest $o'.w$.

After these, TopKCT populates T_e by iterating the following until T_e has k tuples or Q becomes empty (lines 7-15, where $\|\cdot\|$ denotes cardinality). In each iteration, an object o is popped out from Q (line 8). If $o.t$ is verified a candidate target via check, $o.t$ is added to T_e (line 9). TopKCT then expands Q with tuples that differ from previous ones in *only one* attribute (lines 10-15). To do so, it first expands B_i by adding the top value from H_i to the end of B_i if $o.p_i$ already points to the last value of vector B_i, *i.e.,* all the values in B_i have been inspected (line 11). It then generates a new o' by letting $o.t[A_i] := B_i[o.p_i + 1]$, *i.e.,* the value with the highest score next to o (lines 12-14; to simplify the discussion, we assume a weight w_{A_i} for each value when computing $o'.w$, as in Section 6.3; but this can be lifted). If o' has not been pushed to Q before, *i.e.,* $o' \notin T$, o' is added to Q and T (line 15).

Example 6.6.2: Consider the same S and $(2, p(\cdot))$ as in Example 6.6.1. Instead of ranked lists, TopKCT takes as input two heaps H_{team} = {Chicago Bulls; Chicago; Birmingham Barons; $\perp_{\mathsf{team}}$} and H_{arena} = {United Center; Chicago Stadium; Regions Park; $\perp_{\mathsf{arena}}$}. It first pops the top values from H_{team} and H_{arena}, which are Chicago

Bulls and United Center, to form t_0 with $p(t_0)$ = 2+2 = 4. It puts t_0 in a Brodal Queue Q. In the first iteration, TopKCT pops the top tuple out of Q (*i.e.,* t_0), and adds it to T_e since t_0 is a valid candidate target. It then pushes t_1 and t_2 to Q, where $t_1[\mathsf{team}, \mathsf{arena}]$ = (Chicago, United Center) and $t_2[\mathsf{team}, \mathsf{arena}]$ = (Chicago Bulls, Chicago Stadium). Then, t_1 (or t_2) is added to T_e as it is valid and is among the first popped from Q. Finally TopKCT returns $T_e = [t_0, t_1]$. □

Analysis. Algorithm TopKCT generates the next best candidate tuple by changing *one* attribute of some tuple already in T_e. As argued earlier, this strategy suffices to find top-k tuples. Better still, TopKCT has the following properties.

Proposition 6.6.3: TopKCT *has the early termination property,* i.e., *it stops as soon as T_e is found. In addition, it is instance optimal* w.r.t. *the number of visits of each heap* (pop*'s) among all exact algorithms that use heaps to find top-k candidate targets, with optimality ratio 1.* □

An algorithm A is said to be *instance optimal* if there exist constants c_1 and c_2 such that $cost(A,I) \leq c_1 \cdot cost(A',I) + c_2$ for all instances I and all algorithms A' in the same setting as A, where $cost(A,I)$ is a *cost metric* of A on I [FLN03]. The constant c_1 is called the *optimality ratio*. Here $cost(A,I)$ is the number of pop's performed on each heap by A on I.

Proof sketch: TopKCT terminates as soon as $\|T_e\|$ reaches k, without further inspecting other tuples. To see that it is instance optimal, assume by contradiction that there exists an exact algorithm $\mathcal{A}$ that needs two less pop operations. Then we show that there exist two specifications S and S' such that their heaps are the same except in one, they lead to different top-k sets, but $\mathcal{A}$ cannot distinguish the two. □

Complexity. TopKCT incurs less cost than RankJoinCT. To see this, let n be the maximum size of H_i for $i \in [1,m]$, and the kth tuple in T_e correspond to the Kth tuple in the product of domain values. Then (1) pop of a heap takes at most $O(\log\, n)$ time, and there are at most $K + m$ such operations; (2) there are at most K pop operations on the queue Q, and each takes $O(\log\, Km)$ time; (3) there are at most Km push operations on Q, and each takes $O(1)$ time; and (4) check is invoked K times. Denote the cost of invoking check by c. Putting these together, TopKCT is in $O((K+m)\log\, n$ + $K(m$ + $\log K$ + $\log m$ + c)) time, in contrast to exponential in K by RankJoinCT.

In light of the inherent intractability, however, K may be an exponential of n in the worst case, *e.g.,* when S does not have k candidate targets; in this case, TopKCT would inevitably exhaust the entire search space. Nevertheless, one can easily verify the following tractable special cases.

Proposition 6.6.4: TopKCT *is in* PTIME *when (1) ARs are of form (2) only, or (2) the schema R of I_e is fixed.* □

For instance, in case (1) one can easily see that $K = k$; hence from the analysis above it follows that RJCT is in PTIME, whereas TopKCT still takes exponential time.

As will be experimentally verified shortly, TopKCT actually scales well with real-life data. In addition, by modifying the check step for checking Church-Rosser, TopKCT can also be used to *compute top-k rank joins of unranked lists*.

6.6.3 TopKCT$_\mathsf{h}$: A Heuristic Algorithm

Finally we outline TopKCT$_\mathsf{h}$, a PTIME heuristic algorithm when all the attributes of R have an infinite domain.

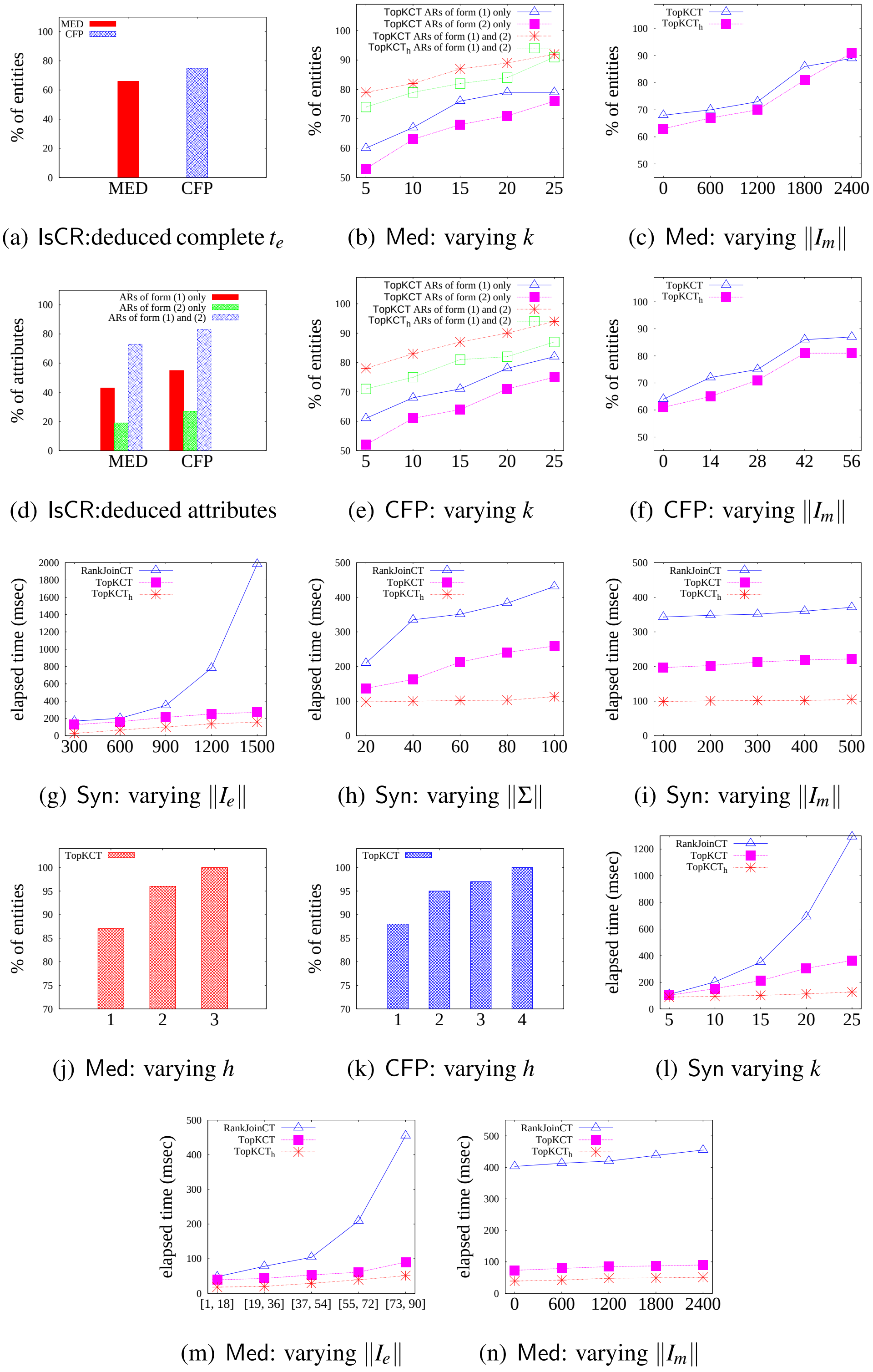
TopKCT$_\mathsf{h}$ first finds a set of k tuples by simply invoking TopKCT without the check step (*i.e.,* line 9 in TopKCT). For each t returned by TopKCT, it greedily revises t with values from I_e and I_m until the revised t is verified a candidate target by check(t,S). It returns the revised tuples as T_e.

TopKCT$_\mathsf{h}$ is in $O((k+m)\log n + k(m+\log k+\log m)+km\mathrm{c})$ time, by the complexity of TopKCT ($k = K$ here), since revising k tuples takes $O(km\mathrm{c})$ time. It is heuristic in nature: while tuples in T_e are guaranteed to be candidate targets of S, they do not necessarily have the highest scores, a tradeoff between the cost and the quality of the solutions.

6.7 Experimental Study

Using both real-life data and synthetic data, we conducted five sets of experiments to evaluate: (1) the effectiveness of algorithm IsCR for deducing target tuples; (2) the effectiveness of RankJoinCT, TopKCT and TopKCT$_\mathsf{h}$ for computing top-k candidate targets; (3) rounds of user interactions; (4) the efficiency of RankJoinCT, TopKCT and TopKCT$_\mathsf{h}$; and (5) the effectiveness of TopKCT when being used for truth discovery, compared with the algorithms of [DBES09b, FGTY13].

Experimental setting. Three real-life datasets (Med, CFP and Rest) and synthetic datasets (Syn) were used.



(a) IsCR:deduced complete t_e

(b) Med: varying k

(c) Med: varying $\|I_m\|$

(d) IsCR:deduced attributes

(e) CFP: varying k

(f) CFP: varying $\|I_m\|$

(g) Syn: varying $\|I_e\|$

(h) Syn: varying $\|\Sigma\|$

(i) Syn: varying $\|I_m\|$

(j) Med: varying h

(k) CFP: varying h

(l) Syn varying k

(m) Med: varying $\|I_e\|$

(n) Med: varying $\|I_m\|$

Figure 6.6: Experimental results

(1) Med was provided by a medicine distribution company (name withheld). It contained sale records of medicines from various stores, specified by a relation schema of

30 attributes such as name, regNo, manufacturer, whose values were not very accurate. Med consisted of 10K tuples for 2.7K entities, where each entity instance ranged from 1 to 83 tuples (4 in average). A set of reference data of 2.4K tuples with 5 attributes was also provided by the company, and we treated it as master data. We manually designed 105 ARs for Med, in which 90 were of form (1) and 15 of form (2).

(2) CFP was extracted from *call for papers/participation* found by Google (*e.g.,* WikiCFP[*], Dbworld[†]). Its attributes included venue, program, and deadline, with values varied in different versions of calls for the same conference. CFP consisted of 100 conferences (entities), with 503 tuples and 22 attributes. The entity instances ranged from 1 to 15 tuples (5 in average). We manually cleaned 55 entries from WikiCFP and treated them as master data, with 17 attributes. We found 43 ARs, with 28 of form (1) and 15 of form (2).

(3) Rest *data*. Rest was the restaurant data used by [DBES09b] from `http://lunadong.com/fusionDataSets.htm` . It consisted of 8 snapshots of 5149 restaurants in Manhattan, with 246K tuples, crawled from 12 Web sources in one-week intervals. Only the true value of a Boolean attribute closed? was to be determined. We found 131 ARs for Rest, all of form (1).

(4) Syn *data*. We generated a master relation I_m and entity instances I_e of 20 attributes by extending relations stat (Table 6.1) and nba (Table 6.2), respectively. The values in I_e and I_m were randomly drawn from the same domains. For its preference model, we assigned random scores to the values in the domains. We also randomly generated a set Σ of 100 ARs, in which 75% were of form (1) and 25% of form (2).

Remark. (1) The specifications for Med, CFP and Rest are Church-Rosser. (2) For preference we counted value occurrences (Section 6.3). (3) We used $k = 15$ by default in Experiments 1, 2 and 3, and $k = 1$ in Exp-5 (for truth discovery). (4) The ARs for each of the datasets have *similar* structures and often share the same LHS. For each attribute there are typically 3-4 ARs, and the large number of ARs comes from the number of attributes. One can also use profiling methods to *automatically discover* ARs [CM08, FL01] (see Section 6.4).

Implementation. We implemented the following, all in Python: (1) our algorithms IsCR, RankJoinCT, TopKCT and $\mathsf{TopKCT_h}$; (2) a naive algorithm voting based on the preference model that only counts value occurrences, without using ARs; (3) the truth

[*] http://www.wikicfp.com/cfp/

[†] http://research.cs.wisc.edu/dbworld/browse.html

discovery algorithm DeduceOrder of [FGTY13] using currency constraints and constant CFDs (see Section 6.1); and (4) a probabilistic-based truth discovery algorithm copyCEF that utilizes a Bayesian model based on quality measures and copy relationships on data sources [DBES09b].

All experiments were conducted on a 64bit Linux Amazon EC2 High-CPU Extra Large Instance with 7 GB of memory, 20 EC2 Compute Units, and 1690 GB of storage. Each experiment was repeated 5 times, and the average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness of IsCR. Using real-life data Med and CFP, we evaluated the quality of target tuples deduced by IsCR: (a) the percentage of target tuples that were complete; and (b) the percentage of non-null attribute values.

Complete target tuples. Figure 6.6(a) shows that for 66% of the entities of Med and 72% of CFP, complete target tuples were automatically deduced by IsCR. That is, by leveraging ARs and master data, complete target tuples could be deduced for over 2/3 of the entities *without user interaction*.

*Non-*null *values*. Figure 6.6(d) reports the average percentage of the attributes in Med and CFP for which the most accurate values were deduced. It shows that IsCR found the most accurate values for 42%, 20%, and 73% (resp. 55%, 27%, and 83%) of the attributes in Med (resp. CFP), when ARs of form (1) only, (2) only, and both forms were used, respectively. This tells us the following: (a) IsCR is able to deduce accurate values for a large percentage of attributes, and (b) ARs of forms (1) and (2) interact with each other; indeed, when ARs of both forms were used, the number of accurate values deduced was larger than the *sum* of its counterparts when ARs of form (1) and (2) were used alone. Moreover, when ARs of form (1) or (2) were used only, no complete targets were deduced for Med and CFP (not shown).

Exp-2: Computing top-*k* candidates. We evaluated the effectiveness of TopKCT (RankJoinCT) and $\mathsf{TopKCT_h}$ using Med and CFP. We manually identified the target tuple for each entity, and tested the percentage of entity instances for which the target tuple was among the top-k candidates found by our algorithms. We report the impact of the choice k, the forms of ARs and the size $|I_m|$ of master data on this. Since RankJoinCT and TopKCT are both exact algorithms, the two behaved the same in this set of experiments.

Impact of k. We report the results in Fig. 6.6(b) (resp. 6.6(e)) when k was varied from

5 to 25 for Med (resp. CFP). As shown there, (a) the larger k is, the more target tuples are covered by the top-k candidates, as expected. (b) To find the target tuples, k does not have to be large. Indeed, when $k = 15$, the targets were found by TopKCT for at least 85% of the entities for both Med and CFP; when $k = 25$, 92% (resp. 94%) were found by TopKCT and 91% (resp. 87%) by $\mathsf{TopKCT_h}$ for Med (resp. CFP). (c) TopKCT did slightly better than $\mathsf{TopKCT_h}$ in the quality of candidates found, while $\mathsf{TopKCT_h}$ is more efficient than TopKCT (see Exp-4).

Impact of ARs. Figures 6.6(b) and 6.6(e) also report the results when Σ consisted of ARs of form (1) only, (2) only, or both. When both forms were used, TopKCT did better than when form (1) or (2) was used alone. In contrast to **Exp-1**, in the latter cases TopKCT could still find many target tuples: for both Med and CFP, it found the targets for 90% of entities when ARs of form (1) or (2) were used only, when $k = 25$.

Impact of $\|I_m\|$. We evaluated the impact of the size of I_m by varying the number of tuples in I_m from 0 to 2400 for Med (resp. 0 to 40 for CFP). As shown in Fig. 6.6(c) for Med (resp. Fig. 6.6(f) for CFP), (a) the larger $\|I_m\|$ is, the better TopKCT and $\mathsf{TopKCT_h}$ perform, *i.e.,* master data helps improve the quality of top-k candidate targets found by our algorithms; and moreover, (b) even when master data is *unavailable* (*i.e.,* $|I_m| = 0$), TopKCT and $\mathsf{TopKCT_h}$ *still work.* Indeed, they were still able to find the target tuples for 63% of Med entities and 64% for CFP, when $k = 15$.

We also tested voting in these settings, and found that voting performed much worse than TopKCT and $\mathsf{TopKCT_h}$. It found no more than 50% of target tuples in all the cases.

Exp-3: User interactions. Using Med and CFP, we simulated user interactions as follows. When the deduced target t_e was incomplete, a single attribute B with $t_e[B] =$ null was randomly picked and assigned its accurate value; IsCR and TopKCT were then invoked on the revised t_e. The process repeated until the top-k candidates returned by TopKCT included the target tuple (manually identified, see **Exp-2**).

The results are reported in Figures 6.6(j) and 6.6(k) for Med and CFP, respectively, in which the x-axis indicates the number h of interaction rounds, and the y-axis shows the percentage of the target tuples found. The results show that few rounds of interactions are needed to deduce the targets for all the entities: at most 3 for Med, and 4 for CFP.

Exp-4: Efficiency. Using Syn and Med, we evaluated the efficiency of IsCR, RankJoinCT, TopKCT and $\mathsf{TopKCT_h}$. To test the impact of I_e, Σ, I_m and k, we set

$(\|I_e\|, \|I_m\|, \|\Sigma\|, k)$ = (900, 300, 60, 15) for Syn, and varied one of the four: $\|I_e\|$ from 300 to 1500, $\|\Sigma\|$ from 20 to 100, $\|I_m\|$ from 100 to 500, and k from 5 to 25, while keeping the other three parameters unchanged. For Med we varied $\|I_e\|$ from [1, 18] to [73, 90] and $\|I_m\|$ from 0 to 2400, while keeping $k = 15$ and $\|\Sigma\| = 105$. We found that IsCR took *at most 10ms* in all these cases and hence, do not report it here.

As shown in Figures 6.6(g) and 6.6(m), on Med and Syn, (a) all three top-k algorithms are efficient (in less than 2s); (b) TopKCT and TopKCT$_\mathsf{h}$ scale well with $\|I_e\|$; and (c) TopKCT outperforms RankJoinCT, and TopKCT$_\mathsf{h}$ does better than TopKCT, verifying the analysis of Section 6.6. They behaved consistently when one of $\|\Sigma\|$, $\|I_m\|$ and k was varied, as shown in Figures 6.6(h), 6.6(i) and 6.6(n), and 6.6(l), respectively. For Syn with $\|I_e\|$ = 1500, $\|I_m\|$ = 300 and $\|\Sigma\|$ = 50, TopKCT$_\mathsf{h}$, TopKCT and RankJoinCT took 159ms, 271ms and 1983ms, respectively. For Med with $\|I_e\|$ in $[73, 90]$ and $\|I_m\|$ = 2400 they took 51ms, 90ms and 455ms, respectively.

Exp-5: Truth Discovery. Besides determining relative accuracy, we also evaluated the effectiveness of our algorithm TopKCT in truth discovery, against algorithms voting, DeduceOrder [FGTY13] and copyCEF [DBES09b] on Med, CFP and Rest. Here we used $k = 1$, to find a single target tuple as the true value, in favor of voting, DeduceOrder and copyCEF.

The algorithms were evaluated as follows. (1) We tested voting on Med, CFP and Rest. The results on Med are similar to those on CFP; thus only the results on CFP and Rest are reported here. (2) DeduceOrder was tested on CFP and Rest. For its rules, we extracted all ARs relevant to data currency as currency constraints, and all constant CFDs that can be expressed as ARs (see Section 6.2), for each dataset. No such rules were found for Med, and hence only results on CFP and Rest are reported here. (3) We evaluated copyCEF on Rest only, because its required information on data sources is unavailable for Med and CFP. Indeed, Med was from a *single source*, and CFP was crawled from blog posts or Web pages for which the source accuracy could not be determined.

On CFP. On CFP, we tested how many true values (targets) were correctly derived for its entities by the algorithms. We found that voting, DeduceOrder and TopKCT deduced 37%, 0% and 70%, respectively. TopKCT performed almost twice better than voting. While DeduceOrder was not able to find the complete true values for any entity of CFP, it correctly derived 31% of attribute values, which are, however, still much lower than the 83% deduced by IsCR (Fig. 6.6(d)).

On Rest. On Rest, we evaluated the effectiveness of voting, DeduceOrder, copyCEF and TopKCT. Rest has only one attribute closed? to be determined. Hence we adopted the recall (r), precision (p) and F-measure (F_1) used in [DBES09b]: $r = \frac{|G \cap R|}{|G|}$, $p = \frac{|G \cap R|}{|R|}$ and $F_1 = \frac{2pr}{p+r}$, where R is the set of restaurants that were concluded to be closed by an algorithm, and G is the set of restaurants that were indeed closed.

As remarked earlier, our method is complementary to the probabilistic-based truth discovery approaches. Moreover, probabilities derived by these approaches can be incorporated into our model. Indeed, when the probabilities of attribute values returned by copyCEF are set as weights in our preference model, copyCEF can be treated as TopKCT with an empty set of ARs. When the weights in the preference model for TopKCT are set by value occurrences, voting is a special case of TopKCT with an empty set of ARs.

As reported in Table 6.4, DeduceOrder achieved 100% precision, but was bad on recall (0.15) and F-measure (0.26). While the F-measure of voting was reasonable (0.74), its precision was not very good (0.62). Algorithm copyCEF performed well with balanced precision (0.76) and recall (0.85), and did better in F-measure (0.8) than voting and DeduceOrder. Better still, ARs help here: with a small number of ARs, TopKCT that took value occurrences (like voting) as its preference outperformed copyCEF and voting on F-measure (0.83), and TopKCT that took the possibilities derived by copyCEF further improved copyCEF on precision (0.81) and recall (0.88), even without user interaction.

Observe the following. (1) DeduceOrder did not do well because there was not much currency and consistency information in CFP and Rest that could be utilized by DeduceOrder. Further, the assumption of [FGTY13] that the data has to be once correct was too strong for CFP and Rest. These further highlight the need for the study of relative accuracy with ARs. (2) Even without ARs, TopKCT can incorporate the source accuracy (copyCEF) and preference (voting), and performs well in truth discovery. (3) In contrast to Rest, many real-life datasets have a number of attributes that are logically correlated. TopKCT works better on such data than on Rest.

Summary. We find the following. (1) IsCR is effective: it is able to correctly and automatically deduce complete target tuples for at least 66% of the entities, and the most accurate values for 73% of the attributes in the real-life data. (2) ARs of form (1) and form (2) interact with each other and are effective in determining relative accuracy. (3) Our top-k algorithms RankJoinCT and TopKCT (resp. TopKCT$_\mathsf{h}$) are capable of finding the target tuples for 93% (resp. 88%) of entities without user interaction, even

Method	Prec	Rec	F-msr
DeduceOrder	1.0	0.15	0.26
voting	0.62	0.92	0.74
copyCEF	0.76	0.85	0.8
TopKCT (preference derived by voting)	0.73	0.95	0.82
TopKCT (preference derived by copyCEF)	0.81	0.88	0.85

Table 6.4: Truth Discovery on Rest

when k is small. (4) Few rounds of user interactions are needed for our framework to deduce complete target tuples (3 for Med and 4 for CFP). (5) Our algorithms are efficient: IsCR takes less than 10ms, and $\mathsf{TopKCT}_\mathsf{h}$ and TopKCT take 159ms and 271ms respectively, on entity instances consisting of 1500 tuples, I_m of 300 tuples and Σ of 50 ARs. (6) Our method is complementary to truth discovery algorithms, and can incorporate derived probabilities into our preference model. With a small number of ARs, TopKCT improves voting, DeduceOrder and copyCEF in truth discovery, with or without user interactions.

Chapter 7

Conclusion and Future Work

In this chapter we summarize the results of this thesis and propose future work.

7.1 Summary

Data quality is the key problem of data management. And there are five central issues of data quality, namely, data consistency, data deduplication, data currency, data accuracy and information completeness. In this thesis we have presented a series of frameworks and models, and developed algorithms for the first four issues and the interactions between these issues.

Incremental detection of inconsistencies in distributed data. We have studied incremental CFD violation detection for distributed data, from complexity to algorithms. We have shown that the problem is NP-complete but is *bounded*. We have also developed *optimal* incremental violation detection algorithms for data partitioned vertically or horizontally, as well as optimization methods. Our experimental results have verified that these yield a promising solution for catching errors in distributed data.

Certain fixes. We have the studied the problem of data monitoring, a special case of data repairing, which is to repair data at the time of data entry. We have developed a framework to compute certain fixes at the point of data entry, by interacting with users, along with its underlying algorithm and optimization techniques. Our experimental results with real-life data have verified the effectiveness, efficiency and scalability of our method. These produce a sound method for data monitoring, based on the algorithms we developed,

Interaction between record matching and data repairing. We have taken a first step toward unifying record matching and data repairing, an important issue that, by and large, has been overlooked. We have proposed a uniform framework for interleaving matching and repairing operations, based on cleaning rules derived from CFDs and MDs. We have established the complexity bounds of several fundamental problems for data cleaning with both matching and repairing. We have also proposed deterministic fixes and reliable fixes, and effective methods to find these fixes based on confidence and entropy. Our experimental results have verified that our techniques substantially improve the quality of fixes generated by repairing and matching taken separately.

Inferring data currency and consistency for Conflict Resolution. We have proposed a model for resolving conflicts in entity instances, based on both data currency and data consistency. We have also identified several problems fundamental to conflict resolution, and established their complexity. Despite the inherent complexity of these problems, we have introduced a framework for conflict resolution, along with practical algorithms supporting the framework. Our experimental study has verified that our methods are effective and efficient using real-life and synthetic data. We contend that these yield a promising approach to resolving conflicts in practice.

Data accuracy. We have proposed a model for determining the relative accuracy of entities in the absence of their true values. We have identified fundamental problems relating to accuracy, and established their complexity. Based on these results, we have introduced a framework for deducing relative accuracy, and provided its underlying algorithms. Our experimental results have verified the effectiveness and efficiency of our methods.

7.2 Future Work

There is naturally much more to be done. We list future research directions, and identify problems to be studied.

Data consistency. For the detection of inconsistencies, we are exploring a new framework to unify multiple integrity constraints. For the detection in distrusted data, first, we are currently experimenting with real-life datasets from different applications, to find out when incremental detection is most effective. Second, we also intend to extend our algorithms to data that is partitioned both vertically and horizontally. Third, we plan to develop MapReduce algorithms for incremental violation detection. Fourth, we are to extend our approach to support constraints defined in terms of similarity predicates (*e.g.,* matching dependencies for record matching) beyond equality comparison, for which hash-based indices may not work and more robust indexing techniques need to be explored.

For the data repairing, our work on certain fixes is just a first step towards repairing data with correctness guarantees. One subject for future work is to efficiently find certain fixes for data in a database, *i.e.,* certain fixes in general data repairing rather than monitoring. Another topic is to develop data repairing and monitoring methods with correctness guarantees in the absence of high-quality master data. Finally, effective algorithms have to be in place for discovering editing rules from sample inputs and master data, along the same lines as discovering other data quality rules [CM08, GKK$^+$08]. Our work on data repairing with interaction with record matching is preliminary. We are also studying the cleaning of multiple relations of which the consistency is specified by constraints across relations, *e.g.,* (conditional) inclusion dependencies. A final subject is to repair data by using currency constraints and partial temporal orders. This is more challenging than conflict resolution, since a database to be repaired is typically *much larger* than entity instances.

Data deduplication. For record matching, we are exploring ways to extend matching dependencies by giving them more expressive power, and integrate the new dependencies into our framework of interaction between record matching and data repairing.

For conflict resolution by inferring data currency and consistency, we are now exploring efficient algorithms with better performance guarantees for generating suggestions, and testing them with data in various domains. Another topic concerns the discovery of data quality rules. Previous work on discovery of such rules [CM08] shows that a large number of high-quality rules can be identified from possibly dirty data. Finally, a challenging topic is to extend our framework by allowing users to edit constraints, and by soliciting other information (such as semantic dependencies specifying how attributes are correlated) as users' feedback when the users do not have sufficient currency knowledge about their data.

The problem of the converse (duplication) of data deduplication problem is worth to be studied. It is common to find that a single record represents two or more external entities in real life. Sometimes splitting the record into multiple records is needed. We are experimenting a new class of integrity constraints and methods to solve this problem.

Data currency. The currency constraints and the framework proposed in Chapter 5 can be applied to infer data currencies. We are exploring ways to simplify the semantics of currency constraints in order to reduce the complexities of algorithms. We are also experimenting with the interaction between data currency and record matching, when tuples/values are from multiple relations with different times.

The principle of parsimony[*] to resolve temporal order is proved to be effective for finding temporal orders in evolution. We are experimenting a new approach based on parsimony, and to see if it could further reduce users' effort when deducing true values.

Data accuracy. The study of data accuracy is still in its infancy. We are currently experimenting with large datasets from other domains to evaluate the techniques. We are also studying how to improve the accuracy of data in a database, which is often much larger than entity instances. Furthermore, discovery of ARs deserves a full treatment.

[*] http://evolution.berkeley.edu/evolibrary/article/phylogenetics_08

Bibliography

[ABC03] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. *TPLP*, 3(4-5), 2003.

[ADNR07] Shipra Agrawal, Supratim Deb, K. V. M. Naidu, and Rajeev Rastogi. Efficient detection of distributed constraint violations. In *ICDE*, 2007.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AKVW93] Alexander Aiken, Dexter Kozen, Moshe Y. Vardi, and Edward L. Wimmers. The complexity of set constraints. In *CSL*, 1993.

[ARS09] Arvind Arasu, Christopher Re, and Dan Suciu. Large-scale deduplication with constraints using Dedupalog. In *ICDE*, 2009.

[Aus99] G. Ausiello. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer, 1999.

[BC81] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1), 1981.

[BCMP10] L. Blanco, V. Crescenzi, P. Merialdo, and P. Papotti. Probabilistic models to reconcile complex data from inaccurate data sources. In *AISE*, 2010.

[BDMW98] James Bailey, Guozhu Dong, Mukesh Mohania, and Xiaoyang Sean Wang. Incremental view maintenance by base relation tagging in distributed databases. *Distributed and Parallel Databases*, 6(3), 1998.

[Ber11] Leopoldo Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

[BESD+09] Laure Berti-Equille, Anish Das Sarma, Xin Dong, A Marian, and Divesh Srivastava. Sailing the information ocean with awareness of currents: Discovery and application of source dependence. In *CIDR*, 2009.

[BFFR05] P Bohannon, W Fan, M Flaster, and R Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[BFG+07] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, 2007.

[BFGM08] Loreto Bravo, Wenfei Fan, Floris Geerts, and Shuai Ma. Increasing the expressivity of conditional functional dependencies without extra complexity. In *ICDE*, 2008.

[BFM07] Loreto Bravo, Wenfei Fan, and Shuai Ma. Extending dependencies with conditions. In *VLDB*, 2007.

[BGMM+09] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BKL11] L. Bertossi, S. Kolahi, and L.V.S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT*, 2011.

[BmLT86] José A. Blakeley, Per Å. Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[BN08] Jens Bleiholder and Felix Naumann. Data fusion. *ACM Comput. Surv.*, 41(1), 2008.

[Bro96] Gerth Stølting Brodal. Worst-case efficient priority queues. In Éva Tardos, editor, *SODA*, pages 52–58. ACM/SIAM, 1996.

[BS06] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[BSIBD09] George Beskales, Mohamed A. Soliman, Ihab F. Ilyas, and Shai Ben-David. Modeling and querying possible repairs in duplicate detection. In *VLDB*, 2009.

[CCC$^+$10] Kuang Chen, Harr Chen, Neil Conway, Joseph M. Hellerstein, and Tapan S. Parikh. Usher: Improving data quality with dynamic forms. In *ICDE*, 2010.

[CFG$^+$07] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.

[CFM09] Wenguang Chen, Wenfei Fan, and Shuai Ma. Analyses and validation of conditional dependencies with built-in predicates. In *DEXA*, 2009.

[CFP84] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *JCSS*, 28(1):29–59, 1984.

[CFY13] Yang Cao, Wenfei Fan, and Wenyuan Yu. Determining the relative accuracy of attributes. In *SIGMOD Conference*, pages 565–576, 2013.

[CGGM03] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[CM05] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, 2005.

[CM08] Fei Chiang and Renee Miller. Discovering data quality rules. In *VLDB*, 2008.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In *Data Base Systems: Courant Computer Science Symposia Series 6*, pages 65–98. Prentice-Hall, 1972.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[CT05] Jan Chomicki and David Toman. Time in database systems. In *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.

[Day83] Umeshwar Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *VLDB*, 1983.

[DBEHS10] Xin Luna Dong, Laure Berti-Equille, Yifan Hu, and Divesh Srivastava. Global detection of complex copying relationships between sources. In *VLDB*, 2010.

[DBES09a] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Integrating conflicting data: The role of source dependence. In *VLDB*, 2009.

[DBES09b] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Truth discovery and copying detection in a dynamic world. In *VLDB*, 2009.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[DHM05] Xin Dong, Alon Y. Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD Conference*, 2005.

[DN09] Xin Luna Dong and Felix Naumann. Data fusion - resolving data conflicts for integration. In *VLDB*, 2009.

[DT07] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *SIGMOD*, 2007.

[dVKCC09] Timothy de Vries, Hui Ke, Sanjay Chawla, and Peter Christen. Robust record linkage blocking using suffix arrays. In *CIKM*, 2009.

[Eck02] Wayne W. Eckerson. Data Quality and the Bottom Line: Achieving Business Success through a Commitment to High Quality Data. In *The Data Warehousing Institute*, 2002.

[EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[Eng00] Larry English. Plain English on data quality: Information quality management: The next frontier. *DM Review Magazine*, April 2000.

[Epp06] M.J. Eppler. *Managing information quality: Increasing the value of information in knowledge-intensive products and processes*. Springer, 2006.

[Fan08] Wenfei Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.

[Fei05] Uriel Feige. Approximating maximum clique by removing subgraphs. *SIAM J. Discret. Math.*, 18, February 2005.

[FG10] W. Fan and F. Geerts. Relative information completeness. *TODS*, 35(4), 2010.

[FG12] Wenfei Fan and Floris Geerts. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, 2012.

[FGJ$^+$11] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. Dynamic constraints for record matching. *VLDB. J*, 20(4):495–520, 2011.

[FGJK08] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(1), 2008.

[FGKT10] Philip Werner Frey, Romulo Goncalves, Martin L. Kersten, and Jens Teubner. A spinning join that does not get dizzy. In *ICDCS*, 2010.

[FGLX11] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *TKDE*, 23(5):683–698, 2011.

[FGMM10] Wenfei Fan, Floris Geerts, Shuai Ma, and Heiko Müller. Detecting inconsistencies in distributed data. In *ICDE*, 2010.

[FGTY13] Wenfei Fan, Floris Geerts, Nan Tang, and Wenyuan Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.

[FGW11] Wenfei Fan, Floris Geerts, and Jef Wijsen. Determining the currency of data. In *PODS*, 2011.

[FH76] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 71(353):17–35, 1976.

[FJLM09] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. In *VLDB*, 2009.

[FL01] Peter A. Flach and Nicolas Lachiche. Confirmation-guided discovery of first-order rules with Tertius. *Machine Learning*, 44(3):61–95, 2001.

[FLM09] C.W. Fisher, E.J.M. Lauria, and C.C. Matheus. An accuracy metric: Percentages, randomness, and probabilities. *JDIQ*, 1(3), 2009.

[FLM$^+$10] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. In *VLDB*, 2010.

[FLM$^+$11a] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.

[FLM$^+$11b] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Cerfix: A system for cleaning data with certain fixes. *PVLDB*, 4(12):1375–1378, 2011.

[FLM$^+$12] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *The VLDB Journal*, 21(2):213–238, April 2012.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[FLR94] C. Fox, A. Levitin, and T. Redman. The notion of data and its quality dimensions. *IPM*, 30(1), 1994.

[FLTY12] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. Incremental detection of inconsistencies in distributed data. In *ICDE*, 2012.

[FLTY14] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. Incremental detection of inconsistencies in distributed data. *TKDE*, 2014.

[FPS$^+$10] Tanveer A. Faruquie, K. Hima Prasad, L. Venkata Subramaniam, Mukesh K. Mohania, Girish Venkatachaliah, Shrinivas Kulkarni, and Pramit Basu. Data cleansing as a transient service. In *ICDE*, 2010.

[GAMS10] A. Galland, S. Abiteboul, A. Marian, and P. Senellart. Corroborating information from disagreeing views. In *WSDM*, 2010.

[GDSZ10] Songtao Guo, Xin Dong, Divesh Srivastava, and Remi Zajac. Record linkage with uniqueness constraints and erroneous values. *PVLDB*, 3(1), 2010.

[Gel10] I.A. Gelman. Setting priorities for data accuracy improvements in satisficing decision-making scenarios: A guiding theory. *DSS*, 48(4), 2010.

[Gil88] P Giles. A model for generalized edit and imputation of survey data. *The Canadian J. of Statistics*, 16:57–73, 1988.

[GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GKK$^+$08] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. In *VLDB*, 2008.

[GM99] Ashish Gupta and Iderpal Singh Mumick. *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[GSTZ03] Sergio Greco, Cristina Sirangelo, Irina Trubitsyna, and Ester Zumpano. Preferred repairs for inconsistent databases. In *IDEAS*, 2003.

[GT04] Enrico Giunchiglia and Armando Tacchella, editors. *Theory and Applications of Satisfiability Testing*, SAT, 2004.

[GW93] Ashish Gupta and Jennifer Widom. Local verification of global integrity constraints in distributed databases. In *SIGMOD*, 1993.

[Ham50] Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[HS98] M. A. Hernandez and S. Stolfo. Real-World Data is Dirty: Data Cleansing and the Merge/Purge Problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[HSW09a] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. *Data Quality and Record Linkage Techniques*. Springer, 2009.

[HSW09b] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. *Data Quality and Record Linkage Techniques*. Springer, 2009.

[Huy97] Nam Huyn. Maintaining global integrity constraints in distributed databases. *Constraints*, 2(3/4), 1997.

[IAE04] I.F. Ilyas, W.G. Aref, and A.K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB. J*, 13(3), 2004.

[IBM12] IBM. What is big data, 2012. `http://www-01.ibm.com/software/data/bigdata/` .

[IBS08] I.F. Ilyas, G. Beskales, and M.A. Soliman. A survey of top-k query processing techniques in relational database systems. *CSUR*, 40(4), 2008.

[Kal08] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 2008.

[KF88] George J. Klir and Tina A Folger. *Fuzzy sets, uncertainty, and information*. Englewood Cliffs, N.J: Prentice Hall, 1988.

[KL09] Solmaz Kolahi and Laks Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.

[KNCV08] Anastasios Kementsietsidis, Frank Neven, Dieter Craen, and Stijn Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. In *VLDB*, 2008.

[Knu09] Donald E. Knuth. *The Art of Computer Programming Volume 4, Fascicle 1: Bitwise tricks & techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[Kos00] D. Kossman. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4), 2000.

[LDK09] Jian Li, Amol Deshpande, and Samir Khuller. Minimizing communication cost in distributed multi-query processing. In *ICDE*, 2009.

[LDMS11] Pei Li, Xin Dong, Andrea Mauricio, and Divesh Srivastava. Linking temporal records. *PVLDB*, 2011.

[Los09] David Loshin. *Master Data Management*. Knowledge Integrity, Inc., 2009.

[MA06] Amihai Motro and Philipp Anokhin. Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. *Inf. Fusion*, 7(2), 2006.

[Ma11] Shuai Ma. *Extending dependencies for improving data quality*. PhD thesis, University of Edinburgh, August 2011.

[MJYE05] Donald W. Miller Jr., John D. Yeast, and Robin L. Evans. Missing prenatal records at a birth center: A communication problem quantified. In *AMIA Annu Symp Proc.*, pages 535–539, 2005.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, 1986.

[MN08] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *SIGMOD*, 2008.

[MNP10] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[NBBW06] Felix Naumann, Alexander Bilke, Jens Bleiholder, and Melanie Weis. Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Eng. Bull.*, 29(2), 2006.

[NH10] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Morgan & Claypool Publishers, 2010.

[NJE+09] P. Narman, P. Johnson, M. Ekstedt, M. Chenine, and J. Konig. Enterprise architecture analysis for data accuracy assessments. In *EDOC*, 2009.

[NPM+10] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. MRShare: Sharing across multiple queries in MapReduce. *PVLDB*, 2010.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.

[Pap94] Christos H Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[PCZ12] D.R.K.P.T.D.D.C.J.G. Paul C. Zikopoulos. *Harness the Power of Big Data: The IBM Big Data Platform*. McGraw-Hill Prof Med/Tech, 2012.

[Red98] Thomas Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM*, 2:79–82, 1998.

[RG95] By Rob and Rob Goldring. Update replication: What every designer should know. In *InfoDB, Vol.9, No.2*, pages 17–24, 1995.

[RH01] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[RN11] Simon Razniewski and Werner Nutt. Completeness of queries over incomplete databases. *PVLDB*, 4(11):749–760, 2011.

[Rou91] Nick Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *TODS*, 16(3), 1991.

[RR93] G. Ramalingam and Thomas W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[RR96] G. Ramalingam and Thomas W. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1&2), 1996.

[RW08] J. Radcliffe and A. White. Key issues for master data management. Technical report, Gartner, 2008.

[SC09] Shaoxu Song and Lei Chen. Discovering matching dependencies. In *CIKM*, 2009.

[SK04] Bart Selman and Henry Kautz. Walksat home page, 2004. *http://www.cs.washington.edu/homes/kautz/walksat/*.

[SMO07] G Sauter, B Mathews, and E Ostic. Information service patterns, part 3: Data cleansing pattern. IBM, 2007.

[SP08] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, 2008.

[SPPN12] Ognjen Savković, Mirza Paramita, Sergey Paramonov, and Werner Nutt. Magik: managing completeness of data. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, CIKM '12, pages 2725–2727, New York, NY, USA, 2012. ACM.

[ST98] Christopher C. Shilakes and Julie Tylman. Enterprise information portals. Technical report, Merrill Lynch, Inc., New York, NY, November 1998.

[Sto76] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theore. Comput. Sci*, 3(1), 1976.

[Sto05] Michael Stonebraker et al. C-store: A column-oriented DBMS. In *VLDB*, 2005.

[SV10] Divesh Srivastava and Suresh Venkatasubramanian. Information theory for data management. In *SIGMOD*, 2010.

[WBGM09] Steven Euijong Whang, Omar Benjelloun, and Hector Garcia-Molina. Generic entity resolution with negative rules. *VLDB J.*, 18(6), 2009.

[WBTD08] Xiaodan Wang, Randal C. Burns, Andreas Terzis, and Amol Deshpande. Network-aware join processing in global-scale database federations. In *ICDE*, 2008.

[WC96] Jennifer Widom and Stefano Ceri. *Active database systems: triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[Wid05] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

[Wij05] Jef Wijsen. Database repairing using updates. *TODS*, 30(3):722–768, 2005.

[Wik12] Wikipedia. Md5, 2012. http://en.wikipedia.org/wiki/MD5 .

[WM11] M. Wu and A. Marian. A framework for corroborating answers from multiple web sources. *IS*, 36(2), 2011.

[WN05] Melanie Weis and Felix Naumann. Dogmatix tracks down duplicates in XML. In *SIGMOD*, 2005.

[WP05] Ingo Wegener and R. Pruim. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005.

[YEN$^+$11] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. Guided data repair. *PVLDB*, 4(1), 2011.

[YENO10] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, and Mourad Ouzzani. GDR: a system for guided data repair. In *SIGMOD*, 2010.

[YHY08] X. Yin, J. Han, and P.S. Yu. Truth discovery with multiple conflicting information providers on the Web. *TKDE*, 20(6), 2008.

[ZDI10] Haopeng Zhang, Yanlei Diao, and Neil Immerman. Recognizing patterns in streams with imprecise timestamps. In *VLDB*, 2010.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE TIT*, 24(5), 1978.

[ZRGH12] Bo Zhao, Benjamin I. P. Rubinstein, Jim Gemmell, and Jiawei Han. A bayesian approach to discovering truth from conflicting sources for data integration. *PVLDB*, 5(6):550–561, 2012.