Extending Dependencies for Improving Data Quality

Shuai Ma

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

This doctoral thesis presents the results of my work on extending dependencies for improving data quality, both in a centralized environment with a single database and in a data exchange and integration environment with multiple databases.

The first part of the thesis proposes five classes of data dependencies, referred to as CINDs, eCFDs, CFD^cs, CFD^ps and CIND^ps, to capture data inconsistencies commonly found in practice in a centralized environment. For each class of these dependencies, we investigate two central problems: the satisfiability problem and the implication problem. The satisfiability problem is to determine given a set Σ of dependencies defined on a database schema $\mathcal{R}$, whether or not there exists a nonempty database D of $\mathcal{R}$ that satisfies Σ. And the implication problem is to determine whether or not a set Σ of dependencies defined on a database schema $\mathcal{R}$ entails another dependency φ on $\mathcal{R}$. That is, for each database D of $\mathcal{R}$ that satisfies Σ, the D must satisfy φ as well. These are important for the validation and optimization of data-cleaning processes. We establish complexity results of the satisfiability problem and the implication problem for all these five classes of dependencies, both in the absence of finite-domain attributes and in the general setting with finite-domain attributes. Moreover, SQL-based techniques are developed to detect data inconsistencies for each class of the proposed dependencies, which can be easily implemented on the top of current database management systems.

The second part of the thesis studies three important topics for data cleaning in a data exchange and integration environment with multiple databases.

One is the dependency propagation problem, which is to determine, given a view defined on data sources and a set of dependencies on the sources, whether another dependency is guaranteed to hold on the view. We investigate dependency propagation for views defined in various fragments of relational algebra, conditional functional dependencies (CFDs) [FGJK08] as view dependencies, and for source dependencies given as either CFDs or traditional functional dependencies (FDs). And we establish lower and upper bounds, all matching, ranging from PTIME to undecidable. These not only provide the first results for CFD propagation, but also extend the classical work of FD propagation by giving new complexity bounds in the presence of a setting with finite domains. We finally provide the first algorithm for computing a minimal cover of all CFDs propagated via SPC views. The algorithm has the same complexity as one of the most efficient algorithms for computing a cover of FDs propagated via a projection view, despite the increased expressive power of CFDs and SPC views.

Another one is matching records from unreliable data sources. A class of matching dependencies (MDs) is introduced for specifying the semantics of unreliable data. As opposed to static constraints for schema design such as FDs, MDs are developed for record matching, and are defined in terms of similarity metrics and a dynamic semantics. We identify a special case of MDs, referred to as relative candidate keys (RCKs), to determine what attributes to compare and how to compare them when matching records across possibly different relations. We also propose a mechanism for inferring MDs with a sound and complete system, a departure from traditional implication analysis, such that when we cannot match records by comparing attributes that contain errors, we may still find matches by using other, more reliable attributes. We finally provide a quadratic time algorithm for inferring MDs, and an effective algorithm for deducing quality RCKs from a given set of MDs.

The last one is finding *certain fixes* for data monitoring [CGGM03, SMO07], which is to find and correct errors in a tuple when it is created, either entered manually or generated by some process. That is, we want to ensure that a tuple t is clean before it is used, to prevent errors introduced by adding t. As noted by [SMO07], it is far less costly to correct a tuple at the point of entry than fixing it afterward.

Data repairing based on integrity constraints may not find *certain fixes* that are absolutely correct, and worse, may introduce new errors when repairing the data. 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 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. We also provide an algorithm to identify minimal certain regions, such that a certain fix is warranted by editing rules and master data as long as one of the regions is correct.

Acknowledgements

I would especially like to thank my supervisor, Professor Wenfei Fan, for too many reasons to be listed here. Without him, I wouldn't have got the chance to study at the University of Edinburgh. He led me into the area of database theory and systems, and taught me the right way to do research. Without his unfailing directions and supports, this thesis wouldn't have been possible. When talking about research, he is strict with me, but he is like a friend in everyday life. I feel that I am really lucky to have him as my supervisor.

I thank Professor Peter Buneman for establishing the database group at Edinburgh, and for creating an active research environment for me to do research. The DB seminar constantly invites great researchers and speakers, which enriches my research.

My special thanks go to Professor Don Sannella. His support helped me get the staff postgraduate scholarship of the University of Edinburgh, which freed me from the burden of tuition fees.

I am very grateful to my collaborators, including Loreto Bravo, Wenguang Chen, Gao Cong, Floris Geerts, Xibei Jia, Jianzhong Li, Jie Liu, Heiko M*ü*ller, Nan Tang, Yinghui Wu, Yunpeng Wu, and Wenyuan Yu. In particular, I thank Floris Geerts for his encouragements, and for spending a lot of his time to discuss questions with me.

I thank the members of my thesis committee: Professor Peter Buneman, Professor Frank Neven and Professor Jef Wijsen, for taking the time to read my thesis and for providing insightful comments and suggestions. I feel honored to have had them on my thesis committee.

I also thank all (past and current) members of the database group at Edinburgh, including Apostolos Apostolidis, Rajendra Bose, Irini Fundulaki, Anastasios Kementsietsidis, Leonid Libkin, Filip Murlak, Anthony Widjaja To, Stratis Viglas, and Xin Wang. It has been a happy time to work around people like them.

Finally, I would like to thank my family: my parents for supporting me with everything that they could provide; and my wife for her love, her support, and for giving up her decent job to come to Edinburgh for staying with me.

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.

(*Shuai Ma*)

Table of Contents

Chapter 1

Introduction

Data cleaning deals with detecting and removing errors and inconsistencies from data in order to improve the quality of data [RD00]. We explore data dependencies based methods for improving data quality, both in a centralized environment with a single database and in a data exchange and integration environment with multiple databases. Data dependencies have been studied for relational databases since the introduction of *functional dependencies* (FDs) by Codd [Cod72] in 1972 (see, *e.g.,* [AHV95, FV84] for details). Recently, data dependencies have generated renewed interests for improving data quality [ABC03b, Ber06, Cho07, Fan08].

A class of *conditional functional dependencies* (CFDs) has recently been proposed in [FGJK08] as an extension of FDs. In contrast to traditional FDs, CFDs hold conditionally on a relation, *i.e.,* they apply only to those tuples that satisfy certain data-value patterns, rather than to the entire relation. CFDs have proven useful in data cleaning [FGJK08, CFG$^+$07, CM08, GKK$^+$08]: inconsistencies and errors in the data may be captured as violations of CFDs, whereas they may not be detected by traditional FDs.

However, CFDs are defined on a single relation, and it has been recognized that to clean data, one needs not only FDs, but also *inclusion dependencies* (INDs) (*e.g.,* [BFFR05, CM05]) defined on two relations. Inspired by the work of CFDs, we propose *conditional inclusion dependencies* (CINDs) in Chapter 2, which extend INDs by enforcing patterns of semantically related data values. Our static analysis of CINDs shows that CINDs retain most nice properties of INDs. For instance, CINDs are always satisfiable and finitely axiomatizable.

Both CFDs and CINDs specify constant patterns in terms of equality ($=$). To capture inconsistencies that commonly arise in real-life data, however, one often needs to use more expressive dependency languages. Motivated by practical examples, we

identify useful constraints to be incorporated into to CFDs and CINDs. The problem is that it usually comes at a price when extending a dependency language with more expressive power. Bearing this in mind, in Chapter 3 we propose three extensions of CFDs: (a) eCFDs which can further express disjunctions and negations, (b) CFD^{c}s which incorporate cardinality constraints and synonym rules, and (c) CFD^{p}s which use $=, \neq, <, \leq, >$ and $\geq$ predicates to specify patterns; and (d) one extension of CINDs, referred to as CIND^{p}s, which specify patterns with $=, \neq, <, \leq, >$ and $\geq$ predicates. The static analysis of these dependencies reveals that the proposed extensions are well balanced between the expressive power and their complexity.

Similar to CFDs, we show how to apply the proposed dependencies, *i.e.,* CINDs, eCFDs, CFD^{c}s, CFD^{p}s and CIND^{p}s, for data cleaning in Chapter 4. We develop SQL-based techniques to detect data inconsistencies for each class of the proposed dependencies. These methods can be easily implemented on the top of current database management systems. This brings immediate benefits for potential users.

So far, we only investigate data quality issues in a centralized environment involving a single database. We then move to a data exchange and integration environment which involves multiple databases, instead of a single one.

We first study the problem that concerns whether the dependencies that hold on data sources still hold on the target data (*i.e.,* data transformed via mapping from the sources), referred to as the *dependency propagation problem*, in Chapter 5. The dependency propagation problem is one of the classical problems in database research, and it is to determine, given a view (mapping) defined on data sources and dependencies that hold on the sources, whether or not another dependency is guaranteed to hold on the view? We refer to the dependencies defined on the sources as *source dependencies*, and those on the view as *view dependencies*. When validating the target database in a data cleaning process, the propagation analysis assures that one need not validate these dependencies that are propagated from the sources via the mapping.

This problem has been extensively studied when source and view dependencies are functional dependencies (FDs), for views defined in relational algebra (*e.g.,* [Fag82, FJT83, Klu80, KP82, Got87]). It is considered an issue already settled in the 1980s. It turns out that while many source FDs may not hold on the view as they are, they do hold on the view under *conditions*. That is, source FDs are indeed propagated to the view, not as standard FDs but as FDs with conditions, *i.e.,* CFDs.

In response to these practical needs, we investigate dependency propagation for views defined in various fragments of relational algebra, CFDs as view dependencies,

and for source dependencies given as either CFDs or traditional functional dependencies (FDs). And we establish lower and upper bounds, all matching, ranging from PTIME to undecidable. These not only provide the first results for CFD propagation, but also extend the classical work of FD propagation by giving new complexity bounds in the presence of a setting with finite domains.

After setting down the complexities, we provide the first algorithm for computing a minimal cover of all CFDs propagated via SPC views. The algorithm has the same complexity as one of the most efficient algorithms for computing a cover of FDs propagated via a projection view [Got87], despite the increased expressive power of CFDs and SPC views.

We then investigates constraints for matching records from *unreliable* data sources in Chapter 6. Record matching is the problem for identifying tuples in one or more relations that refer to the same real-world entity. This problem is also known as record linkage, merge-purge, data deduplication, duplicate detection and object identification. Record matching is a longstanding issue that has been studied for decades. In light of these demands a variety of approaches have been proposed for record matching: probabilistic [FS69, Jar89, Yan07, Win02], learning-based [CR02, SB02, VEH02], distance-based [GKMS04], and rule-based [ACG02, HS95, LSPR96] (see [EIV07] for a recent survey).

The need for record matching is evident. In data integration it is necessary to collate information about an object from multiple data sources [LSPR96]. In data cleaning it is critical to eliminate duplicate records [BS06]. In master data management one often needs to identify links between input tuples and master data [Los09].

No matter what approach to use, one often needs to decide what attributes to compare and how to compare them. Real life data is typically dirty (*e.g.,* a person's name may appear as "Mark Clifford" and "Marx Clifford"), and may not have a uniform representation for the same object in different data sources. To cope with these it is often necessary to hinge on the semantics of the data. Indeed, domain knowledge about the data may tell us what attributes to compare. Moreover, by analyzing the semantics of the data we can deduce alternative attributes to inspect such that when matching cannot be done by comparing attributes that contain errors, we may still find matches by using other, more reliable attributes.

In light of this, we propose a class of *matching dependencies* (MDs) of the form: if some attributes match then *identify* other attributes. In contrast to traditional dependencies, matching dependencies have a *dynamic (update) semantics* to accommodate

errors in unreliable data sources, and they are defined in terms of *similarity operators* and across possibly *different relations*. We then formalize a notion of matching keys, referred to as *relative candidate keys* (RCKs). RCKs are a special class of MDs that match tuples by comparing a *minimum number* of attributes.

To derive RCKs for record matching, we study a generic mechanism for deducing MDs from a given set of MDs, and present an algorithm for deducing RCKs from a set of MDs automatically.

We finally study the problem of finding certain fixes for data monitoring [CGGM03, SMO07], which is to find and correct errors in a tuple when it is created, either entered manually or generated by some process. That is, we want to ensure that a tuple t is clean before it is used, to prevent errors introduced by adding t. As noted by [SMO07], it is far less costly to correct a tuple at the point of entry than fixing it afterward.

A variety of integrity constraints have been studied 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]). These constraints help us determine whether data is dirty or not, *i.e.,* whether errors are present in the data. However, they fall short of telling us which attributes of t are erroneous and moreover, how to correct the errors.

This motivates the quest for effective methods to find *certain fixes* that are guaranteed correct [Gil88, HSW09]. The need for this is evident in monitoring *critical* data, where an error may have disastrous consequences [HSW09]. 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 is being developed by IBM, SAP, Microsoft and Oracle. In particular, master data has been explored to provide *a data entry solution* in the SOA (Service Oriented Architecture) at IBM [SMO07].

To the end, 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. Our approach can be used in data monitoring and enrichment, such that a certain fix is warranted by editing rules and master data as long as one of the certain regions is correct.

We conclude this thesis, and discuss future work in Chapter 8.

1.1 Contributions

The first contribution of the thesis consists of five classes of new data dependencies to capture data inconsistencies commonly found in practice.

- Conditional inclusion dependencies (CINDs) are proposed as an extension of inclusion dependencies (INDs), by enforcing patterns of semantically related data values, in Chapter 2.

  We give a full treatment of the static analysis of CINDs, and show that CINDs retain most nice properties of traditional INDs: (a) CINDs are always satisfiable; (b) CINDs are finitely axiomatizable, *i.e.,* there exists a sound and complete inference system for the implication analysis of CINDs; and (c) the implication problem for CINDs has the same complexity as its traditional counterpart, namely, PSPACE-complete, in the absence of attributes with a finite domain; but it is EXPTIME-complete in the general setting.

  In addition, we investigate two practical fragments of CINDs, namely *acyclic* CINDs and *unary* CINDs. We show the following: (d) in the absence of finite-domain attributes, the implication problem for acyclic CINDs and for unary CINDs retains the same complexity as its traditional counterpart, namely, NP-complete and PTIME, respectively; but in the general setting, it becomes PSPACE-complete and coNP-complete, respectively; and (e) the implication problem for *acyclic unary* CINDs remains in PTIME in the absence of finite-domain attributes and coNP-complete in the general setting.
- Three extensions of conditional functional dependencies (CFDs), referred to as eCFDs, CFD^{c}s, and CFD^{p}s, and one extension of CINDs, referred to as CIND^{p}s, are proposed in Chapter 3. Among these, eCFDs can further express disjunctions and negations; CFD^{c}s can express cardinality constraints, domain-specific conventions, and patterns of semantically related constants in a uniform constraint formalism; and CFD^{p}s and CIND^{p}s specify patterns of data values with $=, \neq, <, \leq, >$ and $\geq$ predicates.

  We show that despite the increased expressive power, eCFDs (resp. CFD^{c}s, CFD^{p}s, and CIND^{p}s) do not make our lives harder. More specially, we show the following: (a) for eCFDs, CFD^{c}s, and CFD^{p}s, their satisfiability problem (resp. their implication problem) remains NP-complete (resp. coNP-complete), the same as their CFD counterparts; and (b) for CIND^{p}s, their satisfiability problem (resp. their implication problem) remains $O(1)$ time (resp. EXPTIME-complete),

the same as their CIND counterparts.

- All the proposed classes of dependencies can be encoded with data tables. And SQL-based methods are developed to detect data inconsistencies in Chapter 4, by treating dependencies and data uniformly.

The second contribution of the thesis investigates dependency propagation for recently proposed conditional functional dependencies (CFDs). The need for this study is evident in data integration, exchange and cleaning since dependencies on data sources often only hold conditionally on the view.

- We investigate dependency propagation for views defined in various fragments of relational algebra, CFDs as view dependencies, and for source dependencies given as either CFDs or traditional functional dependencies (FDs) in Chapter 5.
- We establish lower and upper bounds, all matching, ranging from PTIME to undecidable. These not only provide the first results for CFD propagation, but also extend the classical work of FD propagation by giving new complexity bounds in the presence of a setting with finite domains.
- We provide the first algorithm for computing a minimal cover of all CFDs propagated via SPC views; the algorithm has the same complexity as one of the most efficient algorithms for computing a cover of FDs propagated via a projection view, despite the increased expressive power of CFDs and SPC views.
- We experimentally verify that the algorithm is efficient.

The third contribution of the thesis investigates constraints for matching records from unreliable data sources. A class of matching dependencies (MDs) is introduced for specifying the semantics of unreliable data. As opposed to static constraints for schema design, MDs are developed for record matching, and are defined in terms of similarity metrics and a dynamic semantics.

- We identify a special case of MDs, referred to as relative candidate keys (RCKs), to determine what attributes to compare and how to compare them when matching records across possibly different relations.
- We propose a mechanism for inferring MDs, a departure from traditional implication analysis, such that when we cannot match records by comparing attributes that contain errors, we may still find matches by using other, more reliable attributes.
- We develop a sound and complete system for inferring MDs.
- We provide a quadratic time algorithm for inferring MDs, and an effective algorithm for deducing a set of quality RCKs from MDs.

- We experimentally verify that the algorithms help matching tools efficiently identify keys at compile time for matching, blocking or windowing, and in addition, that the MD-based techniques effectively improve the quality and efficiency of various record matching methods.

The last contribution of the thesis investigates the problem of finding certain fixes for data monitoring. We propose a method for data monitoring, by capitalizing on editing rules and master data.

- We introduce a class of editing rules defined in terms of data patterns and updates in Chapter 7. In contrast to constraints, editing rules have a *dynamic* semantics, and are *relative to* master data.
- We identify and study fundamental problems for reasoning about editing rules. The analyses are relative to a *region* (Z, T_c), where Z is a set of attributes and T_c is a pattern tableau. One problem is to decide whether a set Σ of editing rules guarantees to find a *unique* (deterministic [Gil88, HSW09]) fix for input tuples t that match a pattern in T_c. The other problems concern whether Σ is able to fix all the attributes of such tuples. Intuitively, as long as $t[Z]$ is assured correct, we want to ensure that editing rules can find a certain fix for t. We show that these problems are coNP-complete, NP-complete or #P-complete, but we identify special cases that are in PTIME.
- We develop an algorithm to derive certain regions from a set Σ of rules and master data D_m. A certain region (Z, T_c) is such a region that a certain fix is warranted for an input tuple t as long as $t[Z]$ is assured correct and t matches a pattern in T_c. We naturally want to recommend minimal such Z's to the users. However, we show that the problem for finding minimal certain regions is NP-complete. Nevertheless, we develop an efficient heuristic algorithm to find a set of certain regions, based on a quality model.
- We experimentally verify the effectiveness and scalability of the algorithm.

1.2 Publications

During the course of the PhD study at the University of Edinburgh, I have published the following articles as a co-author.

[FLM+11] Wenfei Fan, Jianzhong Li, **Shuai Ma**, Nan Tang, and Yinghui Wu. Adding Regular Expressions to Graph Reachability and Pattern Queries. In Proceedings of

the 27th International Conference on Data Engineering (ICDE), Hannover, Germany, 2011.

[FGJ+] Wenfei Fan, Hong Gao, Jianzhong Li, Xibei Jia, and **Shuai Ma**. Dynamic Constraints for Record Matching. To appear in The VLDB Journal.

[FLM+10a] Wenfei Fan, Jianzhong Li, **Shuai Ma**, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph Pattern Matching: From Intractable to Polynomial Time. Proceedings of the VLDB Endowment (PVLDB), Volume 3, Singapore, 2010.

[FLM+10b] Wenfei Fan, Jianzhong Li, **Shuai Ma**, Nan Tang, and Wenyun Yu. Towards Certain Fixes with Editing Rules and Master Data. Proceedings of the VLDB Endowment (PVLDB), Volume 3, Singapore, 2010.

[FLM+10c] Wenfei Fan, Jianzhong Li, **Shuai Ma**, Hongzhi Wang, and Yinghui Wu. Graph Homomorphism Revisited for Graph Matching. Proceedings of the VLDB Endowment (PVLDB), Volume 3, Singapore, 2010.

[FGMM10] Wenfei Fan, Floris Geerts, **Shuai Ma**, and Heiko M*ü*ller. Detecting Inconsistencies in Distributed Data. In Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE), California, USA, 2010.

[FJLM09] Wenfei Fan, Xibei Jia, Jianzhong Li, and **Shuai Ma**. Reasoning about Record Matching Rules. Proceedings of the VLDB Endowment (PVLDB), Volume 2, France, 2009.

[CFM09a] Wenguang Chen, Wenfei Fan, and **Shuai Ma**. Analyses and Validation of Conditional Dependencies with Built-in Predicates. In Proceedings of the 20th International Conference on Database and Expert Systems Applications (DEXA), Austria, 2009.

[CFM09b] Wenguang Chen, Wenfei Fan, and **Shuai Ma**. Incorporating Cardinality Constraints and Synonym Rules into Conditional Functional Dependencies. Information Processing Letters, 109(14), 783–789, 2009.

[FMH+08] Wenfei Fan, **Shuai Ma**, Yanli Hu, Jie Liu, and Yinghui Wu. Propagating Functional Dependencies with Conditions. Proceedings of the VLDB Endowment (PVLDB), Volume 1, New Zealand, 2008.

[BFGM08] Loreto Bravo, Wenfei Fan, Floris Geerts, and **Shuai Ma**. Increasing Expressivity of Conditional Functional Dependencies without Extra Charge Complexity. In Proceedings of the 24th International Conference on Data Engineering (ICDE), Cancun, Mexico, 2008.

[CFG+07] Cong Gao, Wenfei Fan, Floris Geerts, Xibei Jia, and **Shuai Ma**. Improving Data Quality: Consistency and Accuracy. In Proceedings of the 33rd International

Conference on Very Large Data Bases (VLDB), Austria, 2007.

[BFM07] Loreto Bravo, Wenfei Fan, and **Shuai Ma**. Extending Dependencies with Conditions. In Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), Austria, 2007.

[CFJM06] Gao Cong, Wenfei Fan, Xibei Jia, and **Shuai Ma**. PRATA: A System for XML Publishing, Integration and View Maintenance (poster paper). In Proceedings of the UK e-Science All Hands Meeting, Nottingham, UK, 2006.

Remark. It is worth mentioning that the (partial) results of this dissertation appear in the above publications: (a) partial results in Chapter 2 appear in [BFM07], (b) most results in Chapters 3 and 4 appear in [BFGM08, CFM09b, CFM09a], and (c) the results in Chapter 5, Chapter 6 and Chapter 7 appear in [FJLM09, FGJ$^+$], [FMH$^+$08] and [FLM$^+$10b], respectively.

Chapter 2

Extending Inclusion Dependencies with Conditions

This chapter introduces a class of conditional inclusion dependencies (CINDs), which extends inclusion dependencies (INDs) by enforcing patterns of semantically related data values. We show that CINDs are useful not only in data cleaning, but are also in contextual schema matching [BEFF06]. We give a full treatment of the static analysis of CINDs, and show that CINDs retain most nice properties of traditional INDs: (a) CINDs are always satisfiable; (b) CINDs are finitely axiomatizable, *i.e.,* there exists a sound and complete inference system for the implication analysis of CINDs; and (c) the implication problem for CINDs has the same complexity as its traditional counterpart, namely, PSPACE-complete, in the absence of attributes with a finite domain; but it is EXPTIME-complete in the general setting. In addition, we investigate two practical fragments of CINDs, namely *acyclic* CINDs and *unary* CINDs. We show the following: (d) in the absence of finite-domain attributes, the implication problem for acyclic CINDs and for unary CINDs retains the same complexity as its traditional counterpart, namely, NP-complete and PTIME, respectively; but in the general setting, it becomes PSPACE-complete and coNP-complete, respectively; and (e) the implication problem for *acyclic unary* CINDs remains in PTIME in the absence of finite-domain attributes and coNP-complete in the general setting.

2.1 Introduction

A class of *conditional functional dependencies* (CFDs) has recently been proposed in [FGJK08] as an extension of *functional dependencies* (FDs). In contrast to tradi-

	asin	title	type	price
t_1:	a23	Snow White	CD	7.99
t_2:	a12	Harry Potter	book	17.99

(a) Example order data

	isbn	title	price	format
t_3:	b32	Harry Potter	17.99	hardcover
t_4:	b65	Snow White	7.99	paperback

(b) Example book data

	isbn	title	price	genre
t_5:	b32	Harry Potter	17.99	country
t_6:	b65	Snow White	7.99	a-book

(c) Example CD data

Figure 2.1: Example instances D_1 of source and target

tional FDs, CFDs hold conditionally on a relation, *i.e.,* they apply only to those tuples that satisfy certain data-value patterns, rather than to the entire relation. CFDs have proven useful in data cleaning [FGJK08, CFG$^+$07, CM08, GKK$^+$08]: inconsistencies and errors in the data may be captured as violations of CFDs, whereas they may not be detected by traditional FDs.

It has been recognized that to clean data, one needs not only FDs, but also *inclusion dependencies* (INDs) (*e.g.,* [BFFR05, CM05]). Furthermore, INDs are commonly used in schema matching systems, *e.g.,* Clio [HHH$^+$05]: INDs associate attributes in a source schema with semantically related attributes in a target schema. Both schema matching and data cleaning highlight the need for extending INDs along the same lines as CFDs, as illustrated by the examples below.

Example 2.1.1: Consider the two (relational) schemas below, referred to as source and target:

source: order(asin : string, title : string, type : string, price : real)
target: book(isbn : string, title : string, price : real, format : string),
CD(id : string, album : string, price : real, genre : string)

The source database contains a single relation order, specifying items of various types such as books, CDs and DVDs, ordered by customers. The target database has two relations, namely, book and CDs, specifying items of books and CDs ordered by customers, respectively. Example source and target instances D_1 are shown in Fig. 2.1.

To find schema mappings from the source to the target, or detect errors across these databases, one might be tempted to use standard INDs such as:

ind_1: order (title, price) $\subseteq$ book (title, price)
ind_2: order (title, price) $\subseteq$ CD (album, price)

These INDs, however, do not make sense: one cannot expect the title and price of each book item in the order table to find a matching CD tuple; it is similar for the CD items in the order table.

Nevertheless, there are indeed inclusion dependencies from the source to the target, as well as on the target database, but only under certain conditions:

cind_1: order (title, price, type = 'book') $\subseteq$ book (title, price)
cind_2: order (title, price, type = 'CD') $\subseteq$ CD (album, price)
cind_3: CD (album, price, genre = 'a-book') $\subseteq$ book (album, price, format = 'audio')

Here constraint cind_1 states that for each order tuple t, if its type is 'book', then there must exist a book tuple t' such that tuples t and t' agree on their title and price attributes; similarly for constraint cind_2. Constraint cind_3 asserts that for each CD tuple t, if its genre is 'a-book' (audio book), then there must be a book tuple t' such that the title and price of t' are identical to the album and price of t, and moreover, the format of t' must be 'audio'.

Constraints cind_1 and cind_2 specify a form of contextual schema matching studied in [BEFF06]. As shown in [BEFF06], contextual schema matching often allows us to derive sensible schema mapping from a source to a target, which cannot be found via schema matching specified with traditional INDs.

In addition, such constraints allow us to detect errors across different relations. For instance, while D_1 of Fig. 2.1 satisfies cind_1 and cind_2, it violates cind_3. Indeed, tuple t_6 in the CD table has an 'a-book' genre, but it cannot find a match in the book table with 'audio' format. The violation suggests that there may exist inconsistencies in the CD and book tables in the target database. Such inconsistencies cannot be detected by traditional INDs. Note that the book tuple t_4 is not a match for t_6: although t_6 and t_4 agree on their album (title) and price attributes, the format of t_4 is 'paperback' rather than 'audio' as required by cind_3. □

Like CFDs, dependencies $\mathsf{cind}_1 - \mathsf{cind}_3$ are required to hold only on a subset of tuples satisfying certain patterns. In other words, they apply only *conditionally* to relations order, CD, and book. These dependencies are specified with constants, and

hence, cannot be expressed as standard INDs. Although such dependencies are needed for both *schema matching* and *data cleaning*, to the best of our knowledge, no previous work has studied these constraints.

Contributions. To this end we introduce an extension of INDs, and investigate the static analysis of these constraints.
(1) Our first contribution is a notion of *conditional inclusion dependencies* (CINDs). A CIND is defined as a pair consisting of an IND $R_1[X] \subseteq R_2[Y]$ and a *pattern tableau*, where the tableau enforces binding of semantically related data values across relations R_1 and R_2. For example, ind_1, ind_2, and $\mathsf{cind}_1 - \mathsf{cind}_3$ given above can all be expressed as CINDs. In particular, traditional INDs are a *special case* of CINDs. This mild extension of INDs captures a fundamental part of the semantics of data, and suffices to express many applications commonly found in data cleaning and schema matching.
(2) Our second contribution consists of complexity results for fundamental problems associated with CINDs, as well as an inference system for reasoning about CINDs.

Given a set of CINDs, the first question one would ask is whether the CINDs are *satisfiable*, *i.e.,* whether they are "dirty" themselves. Indeed, one does not want to enforce the CINDs on a database at run-time but find, after repeated failures, that the CINDs cannot possibly be satisfied by a *nonempty* database. Similarly, one does not want to match schema based on CINDs that do not make sense. The satisfiability analysis helps users develop satisfiable sets of CINDs for data cleaning and schema matching. Another important question in connection with CINDs concerns the *implication* analysis, which is to decide whether a set of CINDs entails another CIND. The implication analysis is useful in reducing redundant CINDs, and hence improving performance when detecting CIND violations in a database, and speeding up the derivation of schema mappings from CINDs [HHH$^+$05].

For traditional INDs, satisfiability is not an issue: any set of INDs is satisfiable. The implication analysis is PSPACE-complete, and furthermore, it is *finitely axiomatizable*: there exists a finite, sound and complete set of axioms.

We show that although CINDs are more expressive than INDs, they retain most nice properties of their traditional counterpart: (a) CINDs are always satisfiable; (b) the implication of CINDs is finitely axiomatizable; (c) in the absence of attributes with a finite domain, the implication problem for CINDs is also PSPACE-complete. However, in the general setting where finite-domain attributes may be present, the problem becomes EXPTIME-complete.

(3) Our third contribution consists of complexity bounds for reasoning about two fragments of CINDs, namely *acyclic* CINDs and *unary* CINDs, which extend *acyclic* INDs and *unary* INDs (see *e.g.,* [AHV95] for details), respectively. Many CINDs found in practice are either acyclic or unary. For instance, $\mathsf{cind}_1 - \mathsf{cind}_3$ we have seen earlier are acyclic CINDs.

We show that in the absence of attributes with a finite domain, the implication problem for acyclic CINDs is NP-complete, and it is in PTIME for unary CINDs. That is, acyclic CINDs (resp. unary CINDs) retain the same complexity as acyclic INDs [CK84] (resp. unary INDs [CKV90]). Nevertheless, we also show that in the general setting, the implication problem becomes PSPACE-complete for acyclic CINDs, and it is coNP-complete for unary CINDs. This tells us that the increased expressive power of CINDs does not come for free. Nevertheless, these complexity bounds are still lower than their counterparts for general CINDs, namely, PSPACE and EXPTIME, respectively. Therefore, when only acyclic or unary CINDs are needed, we do not have to pay the price of the complexity of the full-fledged CINDs.

We show that further constraining acyclic (or unary) CINDs does not make our lives easier. Indeed, the implication problem for *acyclic unary* CINDs remains coNP-complete in the general setting (while in the absence of attributes with finite domains, it is of course still in PTIME).

These results give a full treatment of the fundamental problems associated with CINDs, an extension of INDs that finds applications in schema matching and data cleaning. (1) We show that one can specify any CINDs without worrying about their satisfiability. (2) We develop an inference system that is sound and complete for the implication analysis of CINDs, which provides algorithmic insight into reasoning about CINDs. (3) We present a complete picture of complexity bounds on the implication analysis of CINDs, when finite-domain attributes are present or absent, for general CINDs and for practical fragments (acyclic CINDs and unary CINDs).

We should remark that CINDs do not introduce a new logical formalism. Indeed, in first-order logic, they can be expressed in a form similar to tuple-generating dependencies (TGDs), which have lately generated renewed interests in data exchange (see [Kol05b] for a survey). However, (a) these simple CINDs suffice to capture data consistency and contextual schema matching commonly found in practice, without incurring the complexity of *full-fledged* TGDs (*e.g.,* the undecidability of their implication problem), and (b) no prior work has studied the satisfiability, implication and finite axiomatizability problems for TGDs in the presence of *constants* or attributes with *finite*

domains.

Organization. The remainder of the chapter is organized as follows. We define CINDs in Section 2.2, and investigate their associated satisfiability and implication problems in Section 2.3. We study acyclic CINDs and unary CINDs in Sections 2.4 and 2.5, respectively, followed by related work in Section 2.6.

2.2 Conditional Inclusion Dependencies

A relational database schema $\mathcal{R}$ is a finite collection of relation schemas $(R_1, \ldots, R_n)$, where for each $i \in [1,n]$, R_i is defined over a finite set of attributes, denoted as $\mathsf{attr}(R_i)$. For each attribute $A \in \mathsf{attr}(R_i)$, its domain is specified in R_i, denoted as $\mathsf{dom}(A)$, which is either finite (*e.g.,* bool) or infinite (*e.g.,* string). We use $\mathsf{finattr}(\mathcal{R})$ to denote the set of all the finite-domain attributes that appear in $\mathcal{R}$.

An instance I_i of R_i is a finite set of tuples such that for each $t \in I_i$, $t[A] \in \mathsf{dom}(A)$ for each attribute $A \in \mathsf{attr}(R_i)$. A database instance D of $\mathcal{R}$ is a collection of relation instances $(I_1, \ldots, I_n)$, where I_i is an instance of R_i for $i \in [1,n]$.

Syntax. A conditional inclusion dependency (CIND) ψ is defined as a pair $(R_a[X; X_p] \subseteq R_b[Y; Y_p], T_p)$, where (1) X, X_p and Y, Y_p are lists of attributes in $\mathsf{attr}(R_a)$ and $\mathsf{attr}(R_b)$, respectively, such that X and X_p (resp. Y and Y_p) are disjoint; (2) $R_a[X] \subseteq R_b[Y]$ is a standard IND, referred to as the IND *embedded* in ψ; and (3) T_p is a tableau, called the *pattern tableau* of ψ, defined on all attributes in X_p and Y_p, and for each A in X_p or Y_p and each tuple $t_p \in T_p$, $t_p[A]$ is a constant in $\mathsf{dom}(A)$.

We adopt the following conventions and notations. (1) Let $X = [A_1, \ldots, A_m]$ and $Y = [B_1, \ldots, B_m]$. We require that $\mathsf{dom}(A_i) \subseteq \mathsf{dom}(B_i)$ for each $i \in [1,m]$. (2) If a list Z of attributes occurs in both X_p and Y_p, we use Z_L and Z_R to indicate the occurrence of Z in X_p and Y_p, respectively. (3) When both X_p and Y_p are empty lists, T_p is an empty set $\emptyset$. (4) Abusing set operations, we use $X \cup Y$ to denote the list of all attributes of X and Y, and $X \setminus Y$ to denote the list obtained from list X by removing all the elements in list Y. We denote $X \cup X_p$ as $\mathsf{LHS}(\psi)$ and $Y \cup Y_p$ as $\mathsf{RHS}(\psi)$, and separate the LHS and RHS attributes in a pattern tuple with '$\|$'. (5) We use nil to denote *an empty list*. (6) In addition, we adopt the common assumption that each finite or infinite domain contains at least two elements [GS85], which was used for, *e.g.,* FDs [AHV95].

Example 2.2.1: Constraints ind_1, ind_2, and cind_1–cind_3 given in Examples 2.1.1 can all be expressed as CINDs, as shown in Fig 2.2: ψ_1 and ψ_2 for ind_1 and ind_2, respec-

$\psi_1 = (\mathsf{order}[\mathsf{title}, \mathsf{price}; \mathsf{nil}] \subseteq \mathsf{book}[\mathsf{title}, \mathsf{price}; \mathsf{nil}], T_1 = \emptyset)$

$\psi_2 = (\mathsf{order}[\mathsf{title}, \mathsf{price}; \mathsf{nil}] \subseteq \mathsf{CD}[\mathsf{album}, \mathsf{price}; \mathsf{nil}], T_2 = \emptyset)$

$\psi_3 = (\mathsf{order}[\mathsf{title}, \mathsf{price}; \mathsf{type}] \subseteq \mathsf{book}[\mathsf{title}, \mathsf{price}; \mathsf{nil}], T_3)$

$\psi_4 = (\mathsf{order}[\mathsf{title}, \mathsf{price}; \mathsf{type}] \subseteq \mathsf{CD}[\mathsf{album}, \mathsf{price}; \mathsf{nil}], T_4)$

$\psi_5 = (\mathsf{CD}[\mathsf{album}, \mathsf{price}; \mathsf{genre}] \subseteq \mathsf{book}[\mathsf{title}, \mathsf{price}; \mathsf{format}], T_5)$

T_3:

type	nil
book	

T_4:

type	nil
CD	

T_5:

genre	format
a-book	audio

Figure 2.2: Example CINDs

tively; ψ_3–ψ_5 for cind_1–cind_3, respectively. Observe that ind_1 and ind_2 are standard INDs embedded in ψ_1 and ψ_2, respectively. In ψ_5, X is $[\mathsf{album}, \mathsf{price}]$, Y is $[\mathsf{title}, \mathsf{price}]$, X_p is $[\mathsf{genre}]$, and Y_p is $[\mathsf{format}]$. The standard IND embedded in ψ_5 is $\mathsf{CD}[\mathsf{album}, \mathsf{price}] \subseteq \mathsf{book}[\mathsf{title}, \mathsf{price}]$. □

Semantics. Consider CIND $\psi = (R_a[X; X_p] \subseteq R_b[Y; Y_p], T_p)$. In general, the embedded IND may not hold on the entire R_a relation: it applies only to R_a tuples matching certain pattern tuples in T_p. We say that an R_a (resp. R_b) tuple t_1 (resp. t_2) *matches* a pattern tuple $t_p \in T_p$ if $t_1[X_p] = t_p[X_p]$ (resp. $t_2[Y_p] = t_p[Y_p]$).

An instance (I_a, I_b) of (R_a, R_b) *satisfies* the CIND ψ, denoted by $(I_a, I_b) \models \psi$, if and only if for *each* tuple t_1 in the relation I_a, and for *each* pattern tuple t_p in the pattern tableau T_p, if $t_1[X_p] = t_p[X_p]$, then *there exists* a tuple t_2 in the relation I_b such that $t_1[X] = t_2[Y]$, and moreover, $t_2[Y_p] = t_p[Y_p]$.

That is, if $t_1[X_p]$ matches the pattern $t_p[X_p]$, then the standard IND embedded in ψ and the pattern specified by t_p must be satisfied. These assure the existence of tuple t_2 such that (1) $t_1[X]$ and $t_2[Y]$ are equal, and (2) $t_2[Y_p]$ must match the pattern $t_p[Y_p]$. Note that $t_1[L] = t_p[L]$ if $L = \mathsf{nil}$.

Intuitively, X_p is used to identify the R_a tuples over which ψ is applied. The pattern on Y_p enforces the matching R_b tuples to have certain values in their Y_p attributes.

We say that a database D satisfies a set Σ of CINDs, denoted by $D \models \Sigma$, if $D \models \psi$ for each $\psi \in \Sigma$.

Two sets Σ_1 and Σ_2 of CINDs are *equivalent*, denoted by $\Sigma_1 \equiv \Sigma_2$, if for any instance D, $D \models \Sigma_1$ iff $D \models \Sigma_2$.

Example 2.2.2: The database D_1 given in Fig. 2.1 satisfies CINDs ψ_3 and ψ_4. However, the INDs embedded in these CINDs do not necessarily hold. For example, while ψ_3 is satisfied, ind_1 in ψ_3 is not. The pattern X_p in $\mathsf{LHS}(\psi_3)$ is used to identify the order

tuples on which ψ_3 has to be enforced, namely, those book tuples; similarly for ψ_4.

On the other hand, ψ_5 is *violated* by the database. Indeed, for CD tuple t_6, there exists a pattern tuple t_p in T_5 such that $t_6[\mathsf{genre}] = t_p[\mathsf{genre}]$ = 'a-book' but there is no tuple t in table book such that $t[\mathsf{format}]$ = 'audio', $t[\mathsf{title}] = t_6[\mathsf{title}]$ = 'Snow White', and $t[\mathsf{price}] = t[\mathsf{price}] = 7.99$. Here the genre pattern is to identify CD tuples on which ψ_5 is applicable, while format is a *constraint* on the book tuples that match those CD tuples via the IND embedded in ψ_5. □

Special case. As shown by ψ_1 and ψ_2 in Example 2.2.1, a standard IND $R_a[X] \subseteq R_b[Y]$ is a special case CINDs: it is a CIND $(R_a[X;X_p] \subseteq R_b[Y;Y_p],\ T_p)$, in which both X_p and Y_p are nil, and the pattern tableau T_p is an empty set $\emptyset$. We shall introduce another two special cases of CINDs, namely, acyclic CINDs and unary CINDs, in Sections 2.4 and 2.5, respectively.

Normal form. A CIND $\psi = (R_a[X;X_p] \subseteq R_b[Y;Y_p], T_p)$ is in *normal form* if T_p only consists of a single pattern tuple t_p. We write ψ as $(R_a[X;X_p] \subseteq R_b[Y;Y_p],\ t_p)$.

It is straightforward to verify that a CIND $\psi = (R_a[X;X_p] \subseteq R_b[Y;Y_p], T_p)$ can be expressed as a set $\Sigma_\psi = \{(R_a[X;X_p] \subseteq R_b[Y;Y_p], t_p) \mid t_p \in T_p\}$ of CINDs in the normal form, which is *equivalent* to $\{\psi\}$, *i.e.,* $\{\psi\} \equiv \Sigma_\psi$.

In light of this, in the sequel we shall only consider CINDs in normal form, without loss of generality.

2.3 Reasoning about CINDs

For any constraint language L, there are two fundamental problems associated with it. One is the satisfiability problem, which is to determine whether a given set of constraints in L has conflicts. The other is the implication problem, which is to derive other constraints from a given set of constraints in L. As remarked in Section 7.1, for a constraint language to be effectively used in practice, it is often necessary to be able to answer these two questions at compile time.

One might be tempted to use a constraint language more powerful than CINDs, *e.g.,* full-fledged TGDs extended by allowing constants (data values). The question is whether the language allows us to effectively reason about its constraints. We need a constraint language that is powerful enough to express dependencies commonly found in schema matching and data cleaning, while at the same time well-behaved enough so that its associated decision problems are tractable or, at the very least, decidable [Kol05b]. For full-fledged TGDs, it was known 30 years ago that the implication

problem is *undecidable* even in the *absence* of data values [BV84].

As found in most database textbooks, standard INDs have several nice properties. (a) INDs are always consistent. (b) For INDs, the implication problem is decidable (PSPACE-complete). (c) Better still, INDs are finitely axiomatizable, *i.e.,* there exists a finite inference system that is sound and complete for implication of CINDs. The question is: when data values are introduced into INDs as found in CINDs, does the extension of INDs still have these properties?

It was observed in [BV84] that if TGDs were extended by including data values, their analysis would become more intriguing. Although we are aware of no previous work on the static analyses of TGDs with constants, the study of CFDs [FGJK08] tells us that data values in the pattern tableaux of dependencies would make our lives much harder. Moreover, for the satisfiability and implication problems, we also have to consider the impact of finite-domain attributes, since a finite domain imposes an additional constraint on how a relation can be populated such that the relation observes the data-value patterns and satisfies the dependencies.

In this section we study the satisfiability and implication problems for CINDs. We show that despite that CINDs contain data values and are more expressive than INDs, they retain most of the nice properties of INDs. That is, CINDs properly balance the expressive power and complexity.

Below we first settle the satisfiability problem for CINDs in positive, in Section 2.3.1. We then study the implication analysis of CINDs in Section 2.3.2. More specifically, we develop a sound and complete inference system for CINDs in Section 2.3.2.1, and then establish the complexity of the implication problem in Section 2.3.2.2. Moreover, we also revisit the implication problem for standard INDs in the presence of finite-domain attributes, an issue that has not been studied.

2.3.1 The Satisfiability Analysis

One cannot expect to derive sensible schema matches or clean data from a set of constraints if the constraints are not satisfiable itself. Thus before any run-time computation is conducted, we have to make sure that the constraints are satisfiable, or in other words, make sense themselves.

The *satisfiability problem* for a constraint language L is to determine, given a finite set Σ of constraints in L defined on a database schema $\mathcal{R}$, whether there exists a nonempty instance D of $\mathcal{R}$ such that $D \models \Sigma$.

Traditional FDs and INDs do not contain data values, and any set of FDs or INDs is always satisfiable [FV83]. However, adding data values to constraints may make their satisfiability analysis much harder. Indeed, CFDs, which extend FDs by adding data-value patterns, may be unsatisfiable, as illustrated by the following example taken from [FGJK08].

Example 2.3.1: Consider a relation schema R with $\mathsf{attr}(R) = \{A, B\}$, and the CFDs below defined on R, refining standard FDs $A \to B$ and $B \to A$:

$$\phi_1\colon\ (A \to B, (\mathsf{true} \parallel b_1)), \quad \phi_2\colon\ (A \to B, (\mathsf{false} \parallel b_2)),$$
$$\phi_3\colon\ (B \to A, (b_1 \parallel \mathsf{false})), \quad \phi_4\colon\ (B \to A, (b_2 \parallel \mathsf{true})),$$

where $\mathsf{dom}(A)$ is bool, and b_1, b_2 are two distinct constants in $\mathsf{dom}(B)$. The CFD ϕ_1 (resp. ϕ_2) asserts that for any R tuple t, if $t[A]$ is true (resp. false), then $t[B]$ must be b_1 (resp. b_2). On the other hand, ϕ_3 (resp. ϕ_4) requires that if $t[B]$ is b_1 (resp. b_2), then $t[A]$ must be false (resp. true).

There exists *no* nonempty instance of R that satisfies all these CFDs. Indeed, for any R tuple t, no matter what Boolean value $t[A]$ is, these CFDs together force $t[A]$ to take the other value from the finite domain bool.

Note that if $\mathsf{dom}(A)$ and $\mathsf{dom}(B)$ were infinite, one could find a tuple t such that $t[A]$ is neither true nor false, and $t[B]$ is not b_1 or b_2; then the R instance $\{t\}$ satisfies these CFDs. This tells us that attributes with a finite domain complicate the satisfiability analysis. □

It is known [FGJK08] that the satisfiability problem for CFDs is NP-complete. As opposed to CFDs, the satisfiability analysis of CINDs is as trivial as their standard INDs counterpart, despite the increased expressive power of CINDs.

Theorem 2.3.1 *Any set of* CINDs *is satisfiable.*

Proof: We show that given a set Σ of CINDs over a database schema $\mathcal{R} = (R_1, \ldots, R_n)$, we can always construct a *nonempty* instance D of $\mathcal{R}$ such that $D \models \Sigma$.

We build such a nonempty instance D as follows. First, for each attribute $A \in \mathsf{attr}(R_i)$ (for $i \in [1, n]$), we construct an active domain $\mathsf{adom}(A)$, which consists of a finite set of data values for attribute A from $\mathsf{dom}(A)$. We then populate the instance D by using these active domains.

(1) We start with the construction of the active domains. For each attribute A, we first collect in $\mathsf{adom}(A)$ all those constants that appear in some pattern of A in the CINDs.

We then propagate these constants from $\mathsf{adom}(A)$ to $\mathsf{adom}(B)$ for each attribute B that is connected to A via a CIND of Σ.

More specifically, for each attribute A in some relation of $\mathcal{R}$, we start with $\mathsf{adom}(A) = \emptyset$. For every CIND $(R_i[X;X_p] \subseteq R_j[Y;Y_p],\ t_p)$ in Σ, if $A \in X_p$ or $A \in Y_p$, we include in $\mathsf{adom}(A)$ the constant $t_p[A]$, *i.e.,* we let $\mathsf{adom}(A) = \mathsf{adom}(A) \cup \{t_p[A]\}$. If $\mathsf{dom}(A)$ is still empty after all the CINDs in Σ have been inspected, we let $\mathsf{adom}(A) = \{c\}$ for an arbitrary constant $c \in \mathsf{dom}(A)$.

We propagate these initial values as follows. For each CIND $(R_a[A_1, A_2, \ldots, A_m; X_p] \subseteq R_b[B_1, \ldots, B_m; Y_p], t_p)$ in Σ, we expand $\mathsf{adom}(B_i)$ by letting $\mathsf{adom}(B_i) = \mathsf{adom}(B_i) \cup \mathsf{adom}(A_i)$ for each $i \in [1,m]$. The propagation process is recursively applied to all attributes, and proceeds until no further changes can be made to $\mathsf{adom}(A)$ of any attribute A. Since it starts with a finite set of values for each $\mathsf{adom}(A)$, it is easy to verify that the process always terminates.

(2) We construct D as follows. For each $R_i(A_1, \ldots, A_k) \in \mathcal{R}$, we define $I_i = \mathsf{adom}(A_1) \times \ldots \times \mathsf{adom}(A_k)$, where $\times$ is the Cartesian product operation [Ram98]. And we define $D = (I_1, \ldots, I_n)$.

It is easy to verify that the database instance D is nonempty, finite and furthermore, that $D \models \Sigma$. □

2.3.2 The Implication Analysis

As remarked earlier, the implication analysis allows us to remove redundancies from data quality rules or schema matching, to improve performance. It is also critical in deriving schema mapping from schema matching [BEFF06, HHH+05].

The *implication problem* for a constraint language L is to determine, given a finite set Σ of constraints in L and another ψ of L, all defined on the same database schema $\mathcal{R}$, whether Σ entails ψ, denoted by $\Sigma \models \psi$, *i.e.,* whether for all instances D of $\mathcal{R}$, if $D \models \Sigma$ then $D \models \psi$.

Example 2.3.2: Consider the set of CINDs given in Fig. 2.2. Let Σ be $\{\psi_1, \psi_2, \psi_5\}$. One wants to verify whether or not $\Sigma \models \psi_3$ and $\Sigma \models \psi_4$. If these hold, then CINDs ψ_3 and ψ_4 are redundant; that is, we only need to focus on CINDs in Σ, and ignore CINDs ψ_3 and ψ_4. □

2.3.2.1 An Inference System

As remarked earlier, for standard INDs the implication problem is not only decidable but also finitely axiomatizable. The finite axiomatizability is a property stronger than the decidability since inference rules reveal the essential properties of the constraints, and the former in fact implies the latter [AHV95].

We show that CINDs are also finitely axiomatizable, by providing an inference system for CINDs, denoted by $\mathcal{I}$. Given a finite set Σ of CINDs and another CIND ψ, we denote by $\Sigma \vdash_{\mathcal{I}} \psi$ if ψ is provable from Σ using rules of $\mathcal{I}$. As will be seen shortly, these rules in $\mathcal{I}$ characterize the CIND implication: they are both *sound*, *i.e.,* if $\Sigma \vdash_{\mathcal{I}} \psi$ then $\Sigma \models \psi$, and *complete*, *i.e.,* if $\Sigma \models \psi$ then $\Sigma \vdash_{\mathcal{I}} \psi$.

Recall that for standard INDs, the inference system consists of three rules: reflexivity, projection-permutation and transitivity [AHV95, CFP84]. To cope with the richer semantics of CINDs, the inference system $\mathcal{I}$ is more complicated than the inference system for INDs.

The inference system $\mathcal{I}$ is shown in Fig. 2.3. We briefly illustrate the inference rules in $\mathcal{I}$ as follows.

Intuitively, rules CIND1–CIND3 correspond to the inference rules for INDs. CIND1 is the reflexivity rule. CIND2 shows that the patterns, *i.e.,* X_p and Y_p, can also be permutated, in addition to permutation and projection of the *embedded* IND. CIND3 extends the transitivity rule. It requires not only the RHS of the first CIND to match the LHS of the second CIND, but also their pattern tuples to be matched, *i.e.,* $t_{p_1}[Y_p] = t_{p_2}[Y_p]$.

CIND4 allows us to instantiate attributes in X and their corresponding attributes in Y. Given a CIND $(R_a[X;X_p] \subseteq R_b[Y;Y_p], t_p)$, we can take an attribute A_j from X and its corresponding attribute B_j in Y, assign a data value in $\mathsf{dom}(A_j)$ to them, and move the attribute A_j (resp. B_j) to the pattern tuple X_p (resp. Y_p) of the CIND.

CIND5 enhances the LHS pattern of a CIND by adding an attribute to the pattern X_p. Consider a CIND $(R_a[X;X_p] \subseteq R_b[Y; Y_p], t_p)$. For any attribute $A \in \mathsf{attr}(R_a)$ that is in neither X nor X_p, we can add A to X_p with an arbitrary value from $\mathsf{dom}(A)$. Intuitively, if ψ holds for all data values in $\mathsf{dom}(A)$, then it definitely also holds for a specific value in $\mathsf{dom}(A)$.

CIND6 weakens the RHS pattern of a CIND by removing an attribute from Y_p. If $(R_a[X;X_p] \subseteq R_b[Y; Y_pB], t_p)$ holds, then for each tuple t_a in R_a that satisfies the pattern $t_p[X_p]$, there is a matching tuple t_b in R_b that satisfies the pattern $t_p[Y_pB]$. If an attribute

CIND1: If X is a list of distinct attributes of R, then $(R[X;\mathsf{nil}] \subseteq R[X;\mathsf{nil}], t_p)$, where $t_p = \emptyset$.

CIND2: If $(R_a[A_1,\ldots,A_m;X_p] \subseteq R_b[B_1,\ldots,B_m;Y_p], t_p)$, then $(R_a[A_{i_1},\ldots,A_{i_k};X'_p] \subseteq R_b[B_{i_1},\ldots,B_{i_k};Y'_p], t'_p)$, where (1) $\{i_1,..,i_k\}$ is a list of distinct integers in $\{1,\ldots,m\}$, or $A_{i_1},\ldots,A_{i_k} = B_{i_1},\ldots,B_{i_k} = \mathsf{nil}$; (2) X'_p and Y'_p are permutations of X_p and Y_p, respectively; and (3) $t'_p = t_p[X'_p||Y'_p]$.

CIND3: If $(R_a[X;X_p] \subseteq R_b[Y;Y_p], t_{p_1})$, $(R_b[Y;Y_p] \subseteq R_c[Z;Z_p], t_{p_2})$, and $t_{p_1}[Y_p] = t_{p_2}[Y_p]$, then $(R_a[X;X_p] \subseteq R_c[Z;Z_p], t_{p_3})$, where $t_{p_3}[X_p] = t_{p_1}[X_p]$, and $t_{p_3}[Z_p] = t_{p_2}[Z_p]$.

CIND4: If $(R_a[A_1,\ldots,A_m;X_p] \subseteq R_b[B_1,\ldots,B_m;Y_p], t_p)$, then $(R_a[A_1,\ldots,A_{j-1},A_{j+1},\ldots,A_m;A_j,X_p] \subseteq R_b[B_1,\ldots,B_{j-1},B_{j+1},\ldots,B_m;B_j,Y_p], t'_p)$, where $A_j \in X$, $t'_p[A_j] \in \mathsf{dom}(A_j)$, $t'_p[B_j] = t'_p[A_j]$, and $t'_p[X_p \parallel Y_p] = t_p$.

CIND5: If $(R_a[X;X_p] \subseteq R_b[Y;Y_p], t_p)$, then $(R_a[X;A,X_p] \subseteq R_b[Y;Y_p], t'_p)$, where $A \in \mathsf{attr}(R_a) \setminus (X \cup X_p)$, $t'_p[A] \in \mathsf{dom}(A)$, and $t'_p[X_p||Y_p] = t_p$.

CIND6: If $(R_a[X;X_p] \subseteq R_b[Y;B,Y_p], t_p)$, then $(R_a[X;X_p] \subseteq R_b[Y;Y_p], t'_p)$, where $t'_p = t_p[X_p||Y_p]$.

CIND7: If $(R_a[X;A,X_p] \subseteq R_b[Y;\ Y_p],\ t_{p_i})$ for $i \in [1,n]$, $t_{p_1}[X_p \parallel Y_p] = \ldots = t_{p_n}[X_p \parallel Y_p]$, $A \in \mathsf{finattr}(R_a)$, and $\mathsf{dom}(A) = \{t_{p_1}[A],\ \ldots,\ t_{p_n}[A]\}$, then $(R_a[X;X_p] \subseteq R_b[Y;Y_p], t_p)$, where $t_p[X_p||Y_p] = t_1[X_p||Y_p]$.

CIND8: If $(R_a[X;A,X_p] \subseteq R_b[Y;B,Y_p], t_{p_i})$ such that $t_{p_1}[X_p \parallel Y_p] = \ldots = t_{p_n}[X_p \parallel Y_p]$, $t_{p_i}[A] = t_{p_i}[B]$ for $i \in [1,n]$, $A \in \mathsf{finattr}(R_a)$ and $\mathsf{dom}(A) = \{t_{p_1}[A],\ \ldots,\ t_{p_n}[A]\}$, then $(R_a[A,X;X_p] \subseteq R_b[B,Y;Y_p], t_p)$, where $t_p[X_p||Y_p] = t_1[X_p||Y_p]$.

Figure 2.3: Inference System $\mathcal{I}$ for CINDs

is removed from Y_p, the CIND certainly holds since the same tuple t_b satisfies $t_p[Y_p]$.

Finally, CIND7 and CIND8 are only needed when there are finite domains. CIND7 says that if we have a set of CINDs that are the same except for the value $t_p[A]$ of a finite-domain attribute A, and the union of all those $t_p[A]$ values covers the entire domain $\mathsf{dom}(A)$, then we can replace the set of CINDs by a single CIND in which attribute A is removed from X_p. That is, the presence of A in the LHS pattern has no effect at all, and hence, we can just exclude it from the CINDs.

CIND8 is, in some way, the inverse of CIND4. If CIND4 is used over a CIND ψ to instantiate the values in the pattern tuple for attributes A and B when $t_p[A]$ ranges over all the values of $\mathsf{dom}(A)$, then CIND8 can take all those CINDs and restore ψ. In short, CIND8 merges a set of CINDs if (1) they differ only in the value of $t_{p_i}[A]$, (2) $t_{p_i}[A]$ ranges over all the values in $\mathsf{dom}(A)$, and (3) there is an attribute B in the RHS of each CIND such that $t_{p_i}[A] = t_{p_i}[B]$.

Example 2.3.3: Recall ψ_1 and ψ_3 from Example 2.2.1. From ψ_1, we can derive ψ_3 by using rule CIND5. As another example, we show the effects of finite-domain attributes. Consider a database consisting of three relations $\mathsf{R}_1(\mathsf{ABC})$, $\mathsf{R}_2(\mathsf{EFG})$, and $\mathsf{R}_3(\mathsf{GHK})$ such that $\mathsf{dom}(B) = \{b_1, b_2\}$ is finite and all the other attributes have an infinite domain, *e.g.,* string. Let $\{\psi'_1, \psi'_2, \psi'_3, \psi'\}$ be a set of CINDs, where

$$\begin{aligned}
\psi'_1 &= (R_1[A;B] \subseteq R_2[E;G], (b_1 \parallel g)),\\
\psi'_2 &= (R_1[A;B] \subseteq R_3[H;G], (b_2 \parallel g)),\\
\psi'_3 &= (R_2[E;\mathsf{nil}] \subseteq R_3[H;G], (\mathsf{nil} \parallel g)), \text{ and}\\
\psi' &= (R_1[A;\mathsf{nill}] \subseteq R_3[H;\mathsf{nill}], \emptyset).
\end{aligned}$$

We illustrate how to derive ψ' from $\{\psi'_1, \psi'_2, \psi'_3\}$ step by step:

(1)	$(R_2[E;G] \subseteq R_3[H;G], (g \parallel g))$,	ψ'_3, CIND5
(2)	$(R_1[A;B] \subseteq R_3[H;G], (b_1 \parallel g))$,	(1), ψ'_1, CIND3
(3)	$(R_1[A;\mathsf{nil}] \subseteq R_3[H;G], (\mathsf{nil} \parallel g))$,	(2), ψ'_2, CIND7
(4)	$\psi' = (R_1[A;\mathsf{nil}] \subseteq R_3[H;\mathsf{nil}], \emptyset)$,	(3), CIND6

That is, $\{\psi'_1, \psi'_2, \psi'_3\} \vdash_{\mathcal{I}} \psi'$. □

The soundness and completeness of $\mathcal{I}$. We next show that $\mathcal{I}$ is sound and complete for the implication analysis of CINDs. That is, for any set Σ of CINDs and another CIND φ defined on the same schema $\mathcal{R}$, $\Sigma \models \psi$ if and only if $\Sigma \vdash_{\mathcal{I}} \psi$.

As we have seen from Example 2.3.1, the presence of finite-domain attributes complicates the implication analysis of CFDs. This is also the case for CINDs: when some attributes in Σ or φ have a finite domain, the implication analysis is more intriguing, as indicated by the rules CIND7 and CIND8 in $\mathcal{I}$. In light of this, we distinguish two settings: (1) in the absence of finite-domain attributes, *i.e.,* when none of the attributes in Σ or φ has a finite domain, and (2) the general setting, when some attributes in Σ or φ may have a finite domain. We show that a set of the rules of $\mathcal{I}$ is sound and complete in both settings.

In the absence of finite-domain attributes. This is the setting in which the inference system for standard INDs was developed [CFP84] (see Section 2.6 for a detailed discussion). In this context, our main result about CIND inference is that the inference rules CIND1–CIND6 of $\mathcal{I}$ make a sound and complete inference system for CINDs. For a set Σ of CINDs and another CIND φ, we use $\Sigma \vdash_{\mathcal{I}(1-6)} \varphi$ to denote that φ can be proved from Σ using rules CIND1–CIND6 of $\mathcal{I}$.

Theorem 2.3.2 *In the absence of finite-domain attributes, the inference rules* CIND1–CIND6 *of $\mathcal{I}$ are sound and complete for the implication analysis of* CINDs.

Proof: For the soundness, we show that given a set $\Sigma \cup \{\psi\}$ of CINDs, if $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$ by using CIND1–CIND6, then $\Sigma \models \psi$. This can be readily verified by induction on the length of proofs with CIND1–CIND6, by showing that the application of each of CIND1–CIND6 is sound as illustrated above.

For the completeness, we show that given a set of CINDs $\Sigma \cup \{\psi\}$ over a relational schema $\mathcal{R}$, if $\Sigma \models \psi$ then $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$, *i.e.,* ψ is provable from Σ by using CIND1–CIND6. We assume *w.l.o.g.* that $\mathcal{R} = (R_1, \ldots, R_n)$, and that ψ is $(R_a[A_1, \ldots, A_m; X_p] \subseteq R_b[B_1, \ldots, B_m; Y_p], t_{p_\psi})$.

The proof consists of three parts. (1) We first develop a chase procedure to characterize $\Sigma \models \psi$. The chase process starts with a single-tuple instance of $\mathcal{R}$, and repeatedly adds tuples (one at a time) to the instance by applying CINDs in Σ until no more CINDs can be applied. (2) We then show that the chase process always terminates, and moreover, that if $\Sigma \models \psi$, then the resulting instance satisfies ψ. (3) Based on these, we finally show that if $\Sigma \models \psi$ then $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$.

(1) We first introduce the chase procedure.

We construct a tuple t_a of schema R_a such that (a) $t_a[A_i] = v_i$ for $i \in [1, m]$; (b) $t_a[A_p] = t_{p_\psi}[A_p]$ for each attribute $A_p \in X_p$ of the CIND ψ; and (c) $t_a[B] = v_0$ for the rest of the attributes $B \in \mathsf{attr}(R_a) \setminus (\{A_1, \ldots, A_m\} \cup X_p)$. Here $v_0, v_1, \ldots, v_m$ are $m+1$ distinct variables

The chase process starts with an instance $D_0 = (I_1, \ldots, I_a, \ldots, I_n)$ of $\mathcal{R}$ such that the instance I_a of schema R_a contains the single tuple t_a, and for each $i \in [1, n]$ and $i \neq a$, the instance I_i of schema R_i is empty.

The chase process adds tuples to the database D_0, one at a time, by making use of a chase operation APPLY. More specially, given a CIND $\psi' = (R_i[U; U_p] \subseteq R_j[V; V_p], t_{p_{\psi'}})$ in Σ and an instance D, $\text{APPLY}(D, \psi')$ transforms the instance D into a new one D' by applying CIND ψ' to D as follows.

- For each tuple $t_i \in I_i$ with $t_i[U_p] = t_{p_{\psi'}}[U_p]$, if there exists no tuple $t_j \in I_j$ such that $t_j[V] = t_i[U]$ and $t_j[V_p] = t_{p_{\psi'}}[V_p]$, we say that the CIND ψ' is *applicable* to D. If so then the chase adds a tuple t_j to I_j such that (a) $t_j[V] = t_i[U]$, (b) $t_j[V_p] = t_{p_{\psi'}}[V_p]$, and (c) $t_j[E] = v_0$ for all the other attributes $E \in \mathsf{attr}(R_j) \setminus (V \cup V_p)$.
- If there exists no such tuple t_i for the CIND ψ', the instance D remains the same, *i.e.,* $D = \text{APPLY}(D, \psi')$.

The chase process stops when it reaches an instance D_f such that no more CINDs in Σ are applicable to D_f, *i.e.,* when $D_f = \text{APPLY}(D, \psi')$ for all CINDs ψ' in Σ. We refer to such D_f as *a result of* $\text{APPLY}(D, \Sigma)$.

(2) We next show that the chase process always terminates, and that all result D_f = APPLY(D,Σ) satisfies ψ if $\Sigma \models \psi$.

Observe that in any step of the chase process, the database is defined in terms of a finite set of elements, *i.e.,* $m+1$ variables $\{v_0, v_1, \ldots, v_m\}$, and the constants appearing in the constant patterns of CINDs in $\Sigma \cup \{\psi\}$. There are only finitely many distinct databases with these elements. From this it follows that the chase process always terminates.

Intuitively, the chase process yields a sequence of databases $D_0, D_1, \ldots, D_f$ such that (a) D_0 is the initial instance, and (b) for each $i \in [0, f-1]$, D_{i+1} = APPLY(D_i, ψ') by applying a CIND ψ' in Σ to D_i. In addition, $D_i \subseteq D_{i+1}$ for all $0 \leq i < f$, and APPLY(D_f, ψ') = D_f for all ψ' in Σ. That is, the instance D_f is a fixpoint, denoted as Chase(Σ, ψ).

It is easy to see that Chase$(\Sigma, \psi) \models \Sigma$. Thus if $\Sigma \models \psi$, we also have that Chase(Σ, ψ) $\models \psi$. That is, the initial tuple $t_a \in I_a$ enforces the existence of another tuple $t_b \in I_b$ such that (a) $t_b[B_i] = v_i$ for $i \in [1, m]$, and (b) $t_b[Y_p] = t_{p_\psi}[Y_p]$.

Example 2.3.4: Consider a database D_0 with three relations shown below, and the CINDs ψ'_1 and ψ'_3 given in Example 2.3.3. Let Σ be $\{\psi'_1, \psi'_3\}$ and ψ = $(R_1[A;B] \subseteq R_3[H;G], (b_1 \parallel g))$. The chase process works on the database D_0 as follows.

I_1:	A	B	C
t_1:	v_1	b_1	v_0

I_2:

E	F	G

I_3:

G	H	K

(1) Initial instance D_0

I_1:	A	B	C
t_1:	v_1	b_1	v_0

I_2:

E	F	G
v_1	v_0	g

I_3:

G	H	K

(2) Instance D_1 after executing APPLY(D_0, ψ'_1)

I_1:	A	B	C
t_1:	v_1	b_1	v_0

I_2:

E	F	G
v_1	v_0	g

I_3:

G	H	K
g	v_1	v_0

(3) Final instance D_2 after executing APPLY(D_1, ψ'_3)

Note that by $\Sigma \models \psi$, there exists a tuple t_3 such that $t_3[H] = t_1[A]$ and $t_3[G]$ = 'g'. □

(3) We finally show that if $\Sigma \models \psi$, then $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$. To prove this, it suffices to show the following Claim.

Claim 2.3.3 *Assume that I_j in the chase process contains a tuple t, where (a) for each $E_i \in \mathsf{attr}(R_j)$ ($i \in [1,k]$ for some $k \in [1,m]$), $t[E_i] = v_{j_i}$ and v_{j_i} is in $\{v_1,\ldots,v_m\}$, and (b) $t[Z_p]$ consists of only constants for a list Z_p of distinct attributes in $\mathsf{attr}(R_j)$. Then $\Sigma \vdash_{\mathcal{I}(1-6)} \psi'$, where ψ' is $(R_a[A_{j_1},\ldots,A_{j_k};X_p] \subseteq R_j[E_1,\ldots,E_k;Z_p], t'_p)$, $t'_p[X_p] = t_{p_\psi}[X_p]$ and $t'_p[Z_p] = t[Z_p]$.*

For if it holds, we can conclude that if $\Sigma \models \psi$, then $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$. Indeed, as remarked earlier, $\mathsf{Chase}(\Sigma,\psi) \models \psi$. Since the initial database D_0 contains the tuple t_a, by the definition of chase, there must be a tuple $t_b \in I_b$ in $\mathsf{Chase}(\Sigma,\psi)$ such that (a) $t_b[B_i] = v_i$ for $i \in [1,m]$, and (b) $t_b[Y_p] = t_{p_\psi}[Y_p]$. Hence by Claim 2.3.3, $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$.

We next prove Claim 2.3.3 by induction on the length of the instance sequence generated by the chase process.

Base case. When the length is 1, the sequence consists of the initial instance D_0, which contains a single tuple t_a in the instance I_a of schema R_a. We show that $(R_a[X';X'_p] \subseteq R_a[X';X'_p], (t_a[X'_p] \parallel t_a[X'_p]))$ for all $X' \subseteq \{A_1,\ldots,A_m\}$ and $X'_p \subseteq X_p$, by repeatedly using CIND4 as follows.

(1) $(R_a[X',X'_p;\mathsf{nil}] \subseteq R_a[X',X'_p;\mathsf{nil}],\emptyset)$, CIND1

(2) $(R_a[X';X'_p] \subseteq R_a[X';X'_p], (t_a[X'_p] \parallel t_a[X'_p]))$, CIND4

Inductive case. Assume Claim 2.3.3 for the first $i+1$ instances $D_0,\ldots,D_i$. We show that Claim 2.3.3 also holds on $D_{i+1} = \textsc{apply}(D_i,\psi')$.

If $D_{i+1} = D_i$, the instance D_i is not changed by APPLY, and the claim obviously holds on D_{i+1} since it holds on D_i.

Assume that $D_{i+1} \neq D_i$. Then there must be a single tuple w inserted into the instance I_j of some schema R_j, *i.e.,* $I_j = I_j \cup \{w\}$, as the result of $\textsc{apply}(D_i,\psi')$ for some $\psi' \in \Sigma$. Assume *w.l.o.g.* that $\psi' = (R_i[C_1,\ldots,C_k;U_p] \subseteq R_j[F_1,\ldots,F_k;V_p],\ t_{p_{\psi'}})$. Since ψ' is applicable to D_i, there exists a tuple u in I_i such that $u[U_p] = t_{p_{\psi'}}[U_p]$. By the induction hypothesis, there is a CIND $\psi_u = (R_a[A_{p_1},\ldots,A_{p_h};X_p] \subseteq R_i[C_{p_1},\ldots,C_{p_h};U'_p], t_{p_u})$ such that (a) for each attribute $C \in \mathsf{attr}(R_i)$, if $u[C]$ is a constant, then $C \in U'_p$, and if $u[C]$ is a variable in $\{v_1,\ldots,v_m\}$, then $C \in \{C_{p_1},\ldots,C_{p_h}\}$, (b) $t_{p_u}[U'_p] = u[U'_p]$, and (c) $\Sigma \vdash_{\mathcal{I}(1-6)} \psi_u$. Based on the tuples w,u and the CINDs ψ' and ψ_u, we can derive the following.

(1) $t_{p_{\psi'}}[U'_p] = u[U'_p]$, $t_{p_u}[U_p] = u[U_p]$, and $U_p \subseteq U'_p$.

(2) Let $\{C_{q_1},\ldots,C_{q_g}\} = \{C_1,\ldots,C_k\} \cap \{C_{p_1},\ldots,C_{p_h}\}$, where $0 \leq g \leq \mathsf{min}(k,h)$. Then $\{A_{q_1},\ldots,A_{q_g}\} \subseteq \{A_{p_1},\ldots,A_{p_h}\}$, and for the tuple w, $w[F] \in \{v_1,\ldots,v_m\}$ *iff* $F \in \{F_{q_1},\ldots,F_{q_g}\}$.

(3) For each attribute $F \in \mathtt{attr}(R_j)$, $w[F]$ is a constant *iff* $F \in V_p$ such that $w[F] = t_{p_{\psi'}}[F]$, or $F = F_i$ $(1 \leq i \leq k)$ such that $w[F] = u[F]$ and $C_i \in U_c = U'_p \cap \{C_1, \ldots, C_k\}$. We use V_c to denote the list of attributes in $\mathtt{attr}(R_j)$ that corresponds to the list of attributes U_c of $\mathtt{attr}(R_i)$ in ψ'.

(4) $U_c \cap \{C_{q_1}, \ldots, C_{q_g}\} = \emptyset$ and $U_c \subseteq U'_p$.

We next show that $\Sigma \vdash_{\mathcal{I}(1-6)} \psi_w$, where ψ_w is the CIND $(R_a[A_{p_1}, \ldots, A_{p_{g'}}; X_p] \subseteq R_j[F_{p_1}, \ldots, F_{p_{g'}}; Z'_p], t_{p_{\psi_w}})$. Observe that (a) $\{F_{p_1}, \ldots, F_{p_{g'}}\} \subseteq \{F_{q_1}, \ldots, F_{q_g}\}$ by (2) above, and (b) $Z'_p \subseteq V_p \cup V_c$ by (3). In addition, we can derive the following.

(1) $(R_i[C_{q_1}, \ldots, C_{q_g}; U_c U_p] \subseteq R_j[F_{q_1}, \ldots, F_{q_g}; V_c V_p], t_{p_1})$, where $t_{p_1}[U_p; V_p] = t_{p_{\psi'}}[U_p; V_p]$ and $t_{p_1}[U_c] = t_{p_1}[V_c] = u[U_c] = w[U_c]$. ψ', CIND2, CIND4

(2) $(R_a[A_{q_1}, \ldots, A_{q_g}; X_p] \subseteq R_i[C_{q_1}, \ldots, C_{q_g}; U'_p], t_{p_2})$, where $t_{p_2}[X_p; U'_p] = t_{p_u}[X_p; U'_p]$. ψ_u,CIND2

(3) $(R_i[C_{q_1}, \ldots, C_{q_g}; U'_p] \subseteq R_j[F_{q_1}, \ldots, F_{q_g}; V_c V_p], t_{p_3})$, where $t_{p_3}[U'_p] = t_{p_2}[U'_p]$ and $t_{p_3}[V_c V_p] = t_{p_1}[V_c V_p]$. (1), CIND5

(4) $(R_a[A_{q_1}, \ldots, A_{q_g}; X_p] \subseteq R_i[C_{q_1}, \ldots, C_{q_g}; U_c U_p], t_{p_4})$, where $t_{p_4}[X_p; U_c U_p] = t_{p_2}[X_p; U_c U_p]$. (2), CIND6

(5) $(R_a[A_{q_1}, \ldots, A_{q_g}; X_p] \subseteq R_j[F_{q_1}, \ldots, F_{q_g}; V_c V_p], t_{p_5})$, where $t_{p_5}[X_p] = t_a[X_p]$ and $t_{p_5}[V_c V_p] = t_{p_1}[V_c V_p]$. (2), (3), CIND3 (or alternatively,(1), (4), CIND3)

(6) $(R_a[A_{p_1}, \ldots, A_{p_{g'}}; X_p] \subseteq R_j[F_{p_1}, \ldots, F_{p_{g'}}; V_c V_p], t_{p_6})$, where $t_{p_6}[X_p; V_c V_p] = t_{p_5}[X_p; V_c V_p]$. (5), CIND2

(7) $(R_a[A_{p_1}, \ldots, A_{p_{g'}}; X_p] \subseteq R_j[F_{p_1}, \ldots, F_{p_{g'}}; Z'_p], t_{p_w})$, where $t_{p_w}[X_p] = t_{p_6}[X_p] = t_a[X_p]$ and $t_{p_w}[Z'_p] = t_{p_6}[Z'_p] = w[Z'_p]$. (6), CIND6

This verifies that Claim 2.3.3 holds on D_{i+1}. From this it follows that the inference rules CIND1–CIND6 are complete for the implication of CINDs in the absence of finite-domain attributes. □

Remark. The proof of Theorem 2.3.2 is inspired by the proof for their INDs counterparts given in [CFP84]. Nevertheless, our proof needs to deal with six rules with constant patterns, whereas the inference system for INDs consists of three rules and does not have to consider constant patterns [CFP84].

In the general setting. As indicated by Example 2.3.3, the presence of finite-domain attributes complicates the implication analysis of CINDs. While one can verify that the inference system of [CFP84] is also complete for standard INDs in the presence of finite-domain attributes, the rules CIND1–CIND6 are no longer complete for CIND implication in this setting.

Example 2.3.5: Consider a database schema $\mathcal{R}$ with relations $R_1(ABCD)$,

$R_2(EFGH)$, and $R_3(IJKL)$ such that $\mathsf{dom}(A) = \{a,b,c\}$, $\mathsf{dom}(D) = \{d,e\}$, $\mathsf{dom}(G) = \{d,e,f\}$, $\mathsf{dom}(L) = \{g,h\}$ and all the other attributes have an infinite domain, *e.g.,* string. Consider a set $\Sigma = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$ of CINDs and another CIND ψ, all defined on $\mathcal{R}$, where

$$\begin{aligned}
&\psi_1 = (R_1[BC;AD] \subseteq R_2[EF;GH], (a,d \parallel d,f)),\\
&\psi_2 = (R_1[BC;AD] \subseteq R_3[IJ;K], (b,d \parallel k)),\\
&\psi_3 = (R_3[IJ;K] \subseteq R_2[EF;GH], (k \parallel d,g)),\\
&\psi_4 = (R_1[BC;A] \subseteq R_2[EF;GH], (c \parallel d,h)),\\
&\psi_5 = (R_1[BC;D] \subseteq R_2[EF;GH], (e \parallel e,k)), \text{ and}\\
&\psi = (R_1[BCD;\mathsf{nil}] \subseteq R_2[EFG;\mathsf{nil}], \emptyset)
\end{aligned}$$

Then $\Sigma \not\vdash_{\mathcal{I}(1-6)} \psi$. In other words, CIND1–CIND6 are no longer complete for proving ψ from Σ. In contrast, ψ can be proved from Σ by using CIND1–CIND8, as follows:

(1) $(R_1[BC;AD] \subseteq R_2[EF;GH], (b,d \parallel d,g))$,	ψ_2, ψ_3, CIND3
(2) $(R_1[BC;AD] \subseteq R_2[EF;GH], (c,d \parallel d,h))$,	ψ_4,CIND5
(3) $(R_1[BC;AD] \subseteq R_2[EF;G], (a,d \parallel d))$,	ψ_1, CIND6
(4) $(R_1[BC;AD] \subseteq R_2[EF;G], (b,d \parallel d))$,	(1), CIND6
(5) $(R_1[BC;AD] \subseteq R_2[EF;G], (c,d \parallel d))$,	(2), CIND6
(6) $(R_1[BC;D] \subseteq R_2[EF;G], (d \parallel d))$,	(3),(4),(5), CIND7
(7) $(R_1[BC;D] \subseteq R_2[EF;G], (e \parallel e))$,	ψ_5, CIND6
(8) $\psi = (R_1[BCD;\mathsf{nil}] \subseteq R_2[EFG;\mathsf{nil}], \emptyset)$,	(6),(7), CIND8

That is, $\Sigma \vdash_{\mathcal{I}} \psi$. □

As suggested by the example, we show that CIND1–CIND8 are sound and complete in the general setting.

Theorem 2.3.4 *The inference system $\mathcal{I}$ is sound and complete for implication of* CINDs *in the general case, where finite-domain attributes may be present.*

Proof: The soundness of $\mathcal{I}$ can be verified by induction on the length of $\mathcal{I}$-proofs.

For the completeness of $\mathcal{I}$, consider a set $\Sigma \cup \{\psi\}$ of CINDs defined on a database schema $\mathcal{R}$. We show that if $\Sigma \models \psi$, then $\Sigma \vdash_{\mathcal{I}} \psi$. Assume that $\mathcal{R} = (R_1, \ldots, R_n)$, and that $\psi = (R_a[A_1, \ldots, A_m; X_p] \subseteq R_b[B_1, \ldots, B_m; Y_p], t_{p_\psi})$.

The proof consists of five parts. (1) We first show that it suffices to consider a special form of CINDs. (2) We then develop a chase procedure for the special form of CINDs, and (3) show that the chase process always terminates. (4) In addition, we

establish an important property of the chase procedure, and based on which (5) we show that if $\Sigma \models \psi$ then $\Sigma \vdash_{\mathcal{I}} \psi$ by using CIND1–CIND8.

(1) We first introduce the special form of CINDs.

A CIND $(R_i[U;U_p] \subseteq R_j[V;V_p], t_p)$ is in the special form if (a) the attribute list U *only* consists of attributes with an infinite domain, and (b) the attribute list U_p contains *all* the finite-domain attributes of schema R_i.

As a result, the attribute list V also contains *only* infinite-domain attributes of the schema R_j. However, it is possible that (a) there is an infinite-domain attribute A of schema R_i such that $A \in U_p$, and (b) there exists a finite-domain attribute B of schema R_j such that $B \notin V_p$.

It suffices to consider CINDs in this special form only. Indeed, given a CIND φ, we show that there exists a set Σ_φ of CINDs in the special form such that Σ_φ is equivalent to φ. Better still, Σ_φ can be proved from φ by using rules in $\mathcal{I}$ and vice versa, *i.e.,* $\{\varphi\} \vdash_{\mathcal{I}} \Sigma_\varphi$ and $\Sigma_\varphi \vdash_{\mathcal{I}} \varphi$.

First, consider CIND $\varphi = (R_i[U,A;U_p] \subseteq R_j[V,B;V_p], t_{p_\varphi})$, where attribute A has a finite domain $\mathsf{dom}(A) = \{a_1,\ldots,a_k\}$. For each $l \in [1,k]$, we define a new CIND $\varphi_l = (R_i[U;A,U_p] \subseteq R_j[V;B,Y_p], t_{p_{\varphi_l}})$ such that $t_{p_{\varphi_l}}[A] = t_{p_{\varphi_l}}[B] =$ 'a_l' and $t_{p_{\varphi_l}}[U_p \parallel V_p] = t_{p_\varphi}[U_p \parallel V_p]$. This is justified by CIND4. Let Σ_φ be $\{\varphi_1,\ldots,\varphi_k\}$. It is easy to verify that $\{\varphi\} \equiv \Sigma_\varphi$, *i.e.,* $\Sigma_\varphi \models \{\varphi\}$ and $\{\varphi\} \models \Sigma_\varphi$.

Next, consider CIND $\varphi = (R_i[U;U_p] \subseteq R_j[V;V_p], t_{p_\varphi})$, where there exists an attribute $A \in \mathsf{attr}(R_i) \setminus (U \cup U_p)$ with a finite domain $\mathsf{dom}(A) = \{a_1,\ldots,a_k\}$. For each $l \in [1,k]$, we construct a CIND $\varphi_l = (R_i[U;A,U_p] \subseteq R_j[V;V_p], t_{p_{\varphi_l}})$ such that $t_{p_{\varphi_l}}[A] =$ 'a_l' and $t_{p_{\varphi_l}}[U_p \parallel V_p] = t_{p_\varphi}[U_p \parallel V_p]$. This is justified by CIND5. Let Σ_φ be $\{\varphi_1,\ldots,\varphi_k\}$. Then $\{\varphi\} \equiv \Sigma_\varphi$.

This shows how we can convert each CIND φ in $\Sigma \cup \{\psi\}$ into an equivalent set Σ_φ of CINDs in the special form. In addition, $\{\varphi\} \vdash_{\mathcal{I}} \Sigma_\varphi$ by successive applications of CIND4 and CIND5, and moreover, $\Sigma_\varphi \vdash_{\mathcal{I}} \varphi$ by successive applications of CIND7 and CIND8. Thus, we can assume *w.l.o.g.* that all the CINDs in $\Sigma \cup \{\psi\}$ are in the special form.

(2) We now give the chase procedure for determining whether $\Sigma \models \psi$, which extends the one given in the proof of Theorem 2.3.2 to further deal with finite-domain attributes.

To deal with the interaction between finite domains and constant patterns in the CINDs, the chase process operates on trees as opposed to relations. In such a tree T, (a) $\mathsf{N}_{\mathsf{root}}$ is its root, (b) each node in T is labeled with a tuple t_j and its schema R_i,

denoted by N = '$\mathsf{R_i} : \mathsf{t_j}$', and (c) for each leaf node $\mathsf{N_{leaf}}$ in T, the path from the root $\mathsf{N_{root}}$ to $\mathsf{N_{leaf}}$, denoted by $\text{PATH}(\mathsf{N_{root}}, \mathsf{N_{leaf}})$, encodes an instance of $\mathcal{R}$, such that for each relation schema R in $\mathcal{R}$, the set I_R of tuples of R carried by the nodes on the path is an instance of R.

We now give the details of the chase process.

The chase process starts with a tree T_0 consisting of only a root node $\mathsf{N_{root}}$ = '$\mathsf{R_a} : \mathsf{t_a}$', in which t_a is a tuple of schema R_a such that (a) $t_a[A_i] = v_i$ for $i \in [1,m]$, (b) $t_a[A] = t_{p_\psi}[A]$ for each attribute $A \in X_p$ of CIND ψ, and (c) $t_a[B] = v_0$ for each B in $\mathsf{attr}(R_a) \setminus (\{A_1, \ldots, A_m\} \cup X_p)$, where $v_0, v_1, \ldots, v_m$ are $m+1$ distinct variables. Observe that for each attribute A in $\mathsf{attr}(R_a)$, if $t_a[A]$ is a variable v_i, then A must have an infinite domain by the definition of the special form of CINDs.

The chase process then repeatedly adds nodes to the tree T_0, a set of nodes at a time, by applying a chase operation APPLY_f. To specify APPLY_f, we need the following notion.

A CIND $\psi' = (R_i[U;U_p] \subseteq R_j[V;V_p], t_{p_{\psi'}})$ in Σ is said to be *applicable* to a node N = '$\mathsf{R_i} : \mathsf{t_i}$' if (a) $t_i[U_p] = t_{p_{\psi'}}[U_p]$; (b) there exists no node $\mathsf{N'}$ = '$\mathsf{R_j} : \mathsf{t_j}$' with $t_j[V] = t_i[U]$ and $t_j[V_p] = t_{p_{\psi'}}[V_p]$ on $\text{PATH}(\mathsf{N_{root}}, \mathsf{N})$; and (c) there exists at least a leaf node $\mathsf{N_{leaf}}$ such that there is no such node $\mathsf{N'}$ on $\text{PATH}(\mathsf{N}, \mathsf{N_{leaf}})$. Intuitively, let D denote the database instance represented by $\text{PATH}(\mathsf{N_{root}}, \mathsf{N_{leaf}})$ on which N appears. Then $D \not\models \psi'$, and hence, we need to enforce ψ' on D.

Given a tree T and the CIND ψ', the chase operation $\text{APPLY}_f(T, \psi')$ transforms T into a new tree T' as follows.

- It traverses T starting from its root node $\mathsf{N_{root}}$ in a breadth-first order, and checks whether there exists a node to which the CIND ψ' is applicable.
- If such a node N is found, then new nodes are added to T to make T', as follows.
  (a) Let $S = \{\mathsf{N_{leaf_1}}, \ldots, \mathsf{N_{leaf_k}}\}$ be the set of leaf nodes in T such that for each $i \in [1,k]$, $\text{PATH}(\mathsf{N}, \mathsf{N_{leaf_i}})$ exists and moreover, there exists no node $\mathsf{N'}$ = '$\mathsf{R_j} : \mathsf{t_j}$' with $t_j[V] = t_i[U]$ and $t_j[V_p] = t_{p_{\psi'}}[V_p]$ on $\text{PATH}(\mathsf{N_{root}}, \mathsf{N_{leaf_i}})$.
  (b) Let V_f be the list of all finite-domain attributes in $\mathsf{attr}(R_j) \setminus (V \cup V_p)$. Let $\rho(V_f)$ denote an instantiation of V_f, *i.e.,* for each attribute $C \in V_f$, $\rho(C)$ is a data value drawn from $\mathsf{dom}(C)$.
  (c) For each possible instantiation $\rho(V_f)$, it generates a *new* node $\mathsf{N'_\rho}$ = '$\mathsf{R_j} : \mathsf{t_\rho}$' such that $t_\rho[V] = t_i[U]$, $t_\rho[V_p] = t_{p_{\psi'}}[V_p]$, $t_\rho[V_f] = \rho(V_f)$, and $t_\rho[C] = v_0$ for all the other attributes C in $\mathsf{attr}(R_j)$. Observe that there are only a finite number of instantiations for the attribute list V_f, and for each attribute $C \in \mathsf{attr}(R_j)$, $t_\rho[C]$ is

a constant if the attribute C has a finite domain.

(d) For each leaf $\mathsf{N}_{\mathsf{leaf}_i}$ ($i \in [1,k]$) in S and each new node N'_ρ, it adds N'_ρ as a child of $\mathsf{N}_{\mathsf{leaf}_i}$. That is, ψ' is enforced on the database represented by $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}})$.

Let T' denote the modified tree. Then the same process repeats starting from the root node of T'.

- If there are no nodes to which ψ' is applicable, the tree T remains unchanged, *i.e.,* $T' = \text{APPLY}_f(T, \psi') = T$.

The chase process stops if no CINDs in Σ are applicable to any nodes in T, *i.e.,* $T = \text{APPLY}_f(T, \psi')$ for each ψ' in Σ.

Intuitively, the chase process augments T_0 and generates a sequence $T_0, T_1, \ldots, T_f$ of trees such that (a) for each $l \in [0, f-1]$, $T_{l+1} = \text{APPLY}_f(T_f, \psi')$ for some $\psi' \in \Sigma$, and all the nodes in tree T_l also appear in T_{l+1}, and (b) $T_f = \text{APPLY}_f(T_f, \phi)$ for each ϕ in Σ, *i.e.,* T_f is a fixpoint reached by APPLY_f. We refer to T_f as a *result* of the chase process with Σ and ψ, denoted by $\mathsf{Chase}(\Sigma, \psi)$.

Example 2.3.6: Consider the set $\Sigma \cup \{\psi\}$ of CINDs given in Example 2.3.5. We first transform each CIND into the special form. For instance, for the CIND ψ, we generate a set $\Sigma_\psi = \{\psi'_1, \psi'_2, \psi'_3, \psi'_4, \psi'_5, \psi'_6\}$ of CINDs, where

$$\psi'_1 = (R_1[BC;AD] \subseteq R_2[EF;G], (a, d \parallel d)),$$
$$\psi'_2 = (R_1[BC;AD] \subseteq R_2[EF;G], (b, d \parallel d)),$$
$$\psi'_3 = (R_1[BC;AD] \subseteq R_2[EF;G], (c, d \parallel d)),$$
$$\psi'_4 = (R_1[BC;AD] \subseteq R_2[EF;G], (a, e \parallel e)),$$
$$\psi'_5 = (R_1[BC;AD] \subseteq R_2[EF;G], (b, e \parallel e)),$$
$$\psi'_6 = (R_1[BC;AD] \subseteq R_2[EF;G], (c, e \parallel e)).$$

We show the chase process for Σ and ψ'_2 in Fig. 2.4, where (a) tree T_0 is the initial tree, (b) T_1 is derived by applying ψ_2 to T_0, (c) T_2 is the result of applying $\psi_{3,1}$ to T_1, and (d) T_3 is produced by applying $\psi_{3,2}$ to T_2. Here $\psi_{3,1} = (R_3[IJ;KL] \subseteq R_2[EF;G], (k, g \parallel d, g))$, and $\psi_{3,2} = (R_3[IJ;KL] \subseteq R_2[EF;G], (k, h \parallel d, g))$. These two CINDs are in the special form, derived from ψ_3 by using CIND5.

Observe that (1) $\Sigma \models \psi'_2$, and (2) for each leaf $\mathsf{N}_{\mathsf{leaf}}$ in the result $T_3 = \mathsf{Chase}(\Sigma, \psi'_2)$, there is a node $\mathsf{N} =$ '$\mathsf{R}_2 : \mathsf{t}$' on $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}})$ with $t[EF] = (v_1, v_2)$ and $t[G] =$ 'd'. □

(3) We next verify that the chase process always terminates.

Observe that in each tree T generated in the chase process, $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}})$ from the root $\mathsf{N}_{\mathsf{root}}$ to each leaf $\mathsf{N}_{\mathsf{leaf}}$ of T represents a database instance D of $\mathcal{R}$. In addition,

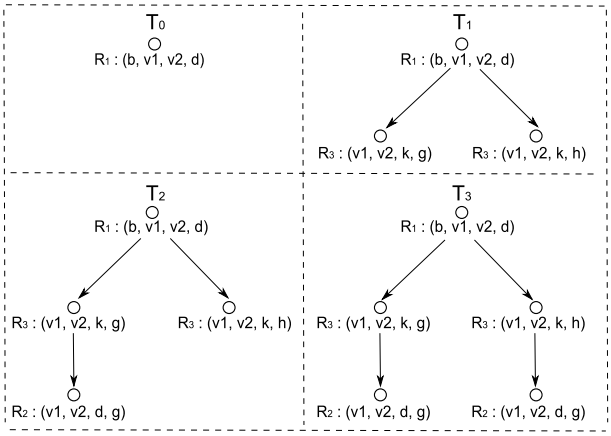


Figure 2.4: An example chasing process

there exist no nodes N_1 and N_2 on the path such that they are labeled with the same tuple. Hence the depth of T is determined by the maximum instance of $\mathcal{R}$ constructed from the finite set $\{v_0, \ldots, v_m\}$ of variables, the finite set of constants appearing in the constant patterns in $\Sigma \cup \{\psi\}$, and all the constants in the finite domains of $\mathcal{R}$. That is, the depth of T is bounded by a constant determined by $\mathcal{R}$ and $\Sigma \cup \{\psi\}$. Similarly, the maximum number of the children of a node in T is bounded by the maximum cardinality of finite domains, which is a constant determined by $\mathcal{R}$. Hence the size of T is bounded by a constant. There exist finitely many distinct trees that are constructed from those variables and constants, with a size bounded by the constant. As a result, the chase process can generate at most finitely many such trees that are distinct, and hence, it must terminate.

(4) We next show a property of the chase procedure, which will be used to show that if $\Sigma \models \psi$, then $\Sigma \vdash_{\mathcal{I}} \psi$.

We first define an operator $\Upsilon(\mathsf{N})$, where N is a node in a tree T_l ($l \in [0, f]$) generated in the chase process. Given N = '$\mathsf{R_i} : \mathsf{t_i}$', we define $\Upsilon(\mathsf{N}) = (R_i[C_1, \ldots, C_m; U_p], t_i[U_p])$ if

- for each $j \in [1, m]$, $t_i[C_j] = v_j$, *i.e.,* $t_i[C_1, \ldots, C_m] = (v_1, \ldots, v_m)$; and
- the list U_p consists of all those attributes $C \in \mathtt{attr}(R_i)$ such that $t_i[C]$ is a constant;

whereas $\Upsilon(\mathsf{N})$ is *undefined* if there exist no attributes $C_1, \ldots, C_m$ in R_i such that $t_i[C_1, \ldots, C_m] = (v_1, \ldots, v_m)$. Observe that when the CIND ψ is enforced, $\Upsilon(\mathsf{N})$ must be defined on some node. We shall use $\Upsilon(\mathsf{N})$ to inspect the existence of nodes satisfying the conditions specified by ψ.

The property is stated as follows.

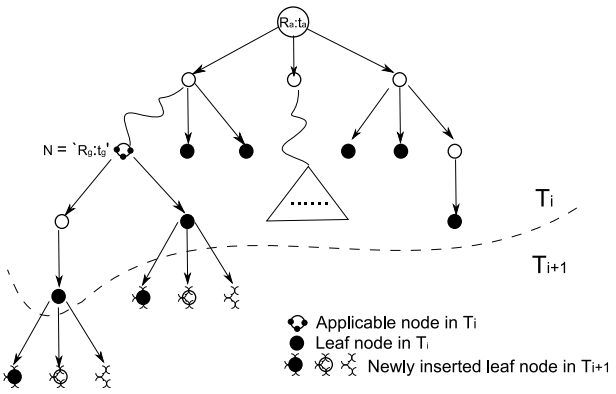


Figure 2.5: Example trees in the chase process

Claim 2.3.5 *Let* φ *be a* CIND $(R_a[A_1,\ldots,A_m;X_p] \subseteq R_{b'}[D_1,\ldots,D_m;V_p],t_{p_\varphi})$*,* T_l *be a tree generated in the chase process (*$l \in [0,f]$*), and S be the set of all the leaf nodes in T, where* $S = \{\mathsf{N}_{\mathsf{leaf}_1},\ldots,\mathsf{N}_{\mathsf{leaf}_k}\}$*. Then* $\Sigma \vdash_{\mathcal{I}} \varphi$ *if* $\Sigma \vdash_{\mathcal{I}} \varphi_j$ *for each* $j \in [1,k]$*, where for a schema* R_i *in* $\mathcal{R}$*,*
(a) $\varphi_j = (R_i[C_{1_j},\ldots,C_{m_j};U_{p_j}] \subseteq R_{b'}[D_1,\ldots,D_m;V_p],t_{p_{\varphi_j}})$ *with* $t_{p_{\varphi_j}}[V_p] = t_{p_\varphi}[V_p]$*, and (b) there exists a node* $\mathsf{N_j}$ = '$\mathsf{R_i : t_j}$' *on* PATH$(\mathsf{N_{root}},\mathsf{N_{leaf_j}})$ *such that* $\Upsilon(\mathsf{N_j}) = (R_i[C_{1_j},\ldots,C_{m_j};U_{p_j}],t_j[U_{p_j}])$ *and* $t_j[U_{p_j}] = t_{p_{\varphi_j}}[U_{p_j}]$*.*

Observe the following. (a) $\mathsf{LHS}(\varphi)$ is the same as $\mathsf{LHS}(\psi)$. (b) For each $i \in [1,k]$, $\mathsf{RHS}(\varphi_j)$ is the same as $\mathsf{RHS}(\varphi)$.As will be seen in part (5) of the proof, we use these to prove $\Sigma \vdash_{\mathcal{I}} \psi$.

We show the claim by induction on the length of the sequence of trees $T_0,T_1,\ldots,T_f$ generated by the chase process.

Base case. For the initial tree T_0, the only leaf node in T_0 is the root $\mathsf{N_{root}}$ = '$\mathsf{R_a : t_a}$'. In this case, $\Upsilon(\mathsf{N_{root}}) = (R_a[A_1,\ldots,A_m;X_p],t_{p_\psi}[X_p])$, and φ_{root} and φ are the same CIND $(R_a[A_1,\ldots,A_m;X_p] \subseteq R_a[A_1,\ldots,A_m;X_p], (t_{p_\psi}[X_p] \parallel t_{p_\psi}[X_p]))$. It is obvious that if $\Sigma \vdash_{\mathcal{I}} \varphi_{root}$, then $\Sigma \vdash_{\mathcal{I}} \varphi$.

Inductive case. Assume that Claim 2.3.5 holds for the first i trees $T_0,T_1,\ldots,T_i$. We show that Claim 2.3.5 also holds on T_{i+1} = APPLY(T_i,ψ'), *i.e.,* a result of applying some CIND ψ' in Σ to a node N in tree T_i. Assume *w.l.o.g.* that N = '$\mathsf{R_g : t_g}$', and that $\psi' = (R_g[U_1;U_{p_1}] \subseteq R_{g'}[U_2;U_{p_2}],t_{p_{\psi'}})$.

We consider two cases: $T_{i+1} = T_i$ and $T_{i+1} \neq T_i$.

If $T_{i+1} = T_i$, the tree T_i is not changed by APPLY(T_i,ψ'), and the claim obviously holds on T_{i+1} since it holds on T_i.

We next focus on the case where $T_{i+1} \neq T_i$. Recall the chase operation

APPLY(T_i, ψ'), by applying the CIND ψ' to the node N. Let $S_i = \{\mathsf{N}_{\mathsf{leaf}_1}, \ldots, \mathsf{N}_{\mathsf{leaf}_k}\}$ be the set of all leaf nodes in tree T_i, and let $S_{new} = \{\mathsf{N}_{\mathsf{f}_1}, \ldots, \mathsf{N}_{\mathsf{f}_h}\}$ be the set of newly generated nodes by applying the CIND ψ' to the node N. In T_{i+1}, all the nodes in S_{new} appear as the children of each leaf node of the sub-tree rooted at N in tree T_i.

To illustrate this, an example of T_i and T_{i+1} is shown in Fig. 2.5. In T_i, the sub-tree rooted at node N = '$\mathsf{R_g} : \mathsf{t_g}$' has two leaf nodes. In T_{i+1}, three new nodes are added as the children of each of the two leaf nodes.

To simplify the discussion we assume *w.l.o.g.* that there is a single leaf node $\mathsf{N}_{\mathsf{leaf}_1}$ in the sub-tree rooted at node N; the proof for multiple such leaf nodes is similar. Thus, the set S_{i+1} of leaf nodes in tree T_{i+1} becomes $S_i \cup S_{new} = \{\mathsf{N}_{\mathsf{f}_1}, \ldots, \mathsf{N}_{\mathsf{f}_h}, \mathsf{N}_{\mathsf{leaf}_2}, \ldots, \mathsf{N}_{\mathsf{leaf}_k}\}$.

We show that the claim holds on T_{i+1}, by considering the following cases.

Case 1. When the operator $\Upsilon(\mathsf{N})$ is undefined on the node N. Then for each node $\mathsf{N}_{\mathsf{f_j}}$ in S_{new} ($j \in [1,h]$), $\Upsilon(\mathsf{N}_{\mathsf{f_j}})$ is also undefined by the definition of APPLY$_f$, which generated those nodes in S_{new} to enforce ψ'. In this case, the claim holds on T_{i+1}. Indeed, those nodes $\mathsf{N_j}$ ($j \in [1, k+h-1]$) required by the claim are in the tree T_i, and so is N. Hence $\Sigma \vdash_{\mathcal{I}} \varphi$ since the claim holds on T_i by the induction hypothesis.

Case 2. When $\Upsilon(\mathsf{N})$ is defined on N. Consider $\Upsilon(\mathsf{N}) = (R_g[U'; U'_p], t_g[U'_p])$. Since the CIND ψ' is applicable to the node N, we can derive the following. (a) $U_{p_1} \subseteq U'_p$, (b) $t_g[U_{p_1}] = t_{p_{\psi'}}[U_{p_1}]$, (c) $t_g[U'] = (v_1, \ldots, v_m)$, and (d) $t_g[C] = v_0$ for each attribute C of $\mathsf{attr}(R_g)$ that is not in $U' \cup U'_p$. We distinguish the following cases.

Case 2(a). $U' \not\subseteq U_1$, where U_1 is in $\mathsf{LHS}(\psi'_1)$. By $\Upsilon(\mathsf{N})$ we have that $t_g[U'] = (v_1, \ldots, v_m)$. Thus, if $U' \not\subseteq U_1$, Υ is not defined on those new nodes in S_{new}. Along the same lines as for Case 1 above, one can show that the claim holds on T_{i+1}.

Case 2(b). $U' \subseteq U_1$. We show that $\Sigma \vdash_{\mathcal{I}} \varphi$ if $\Sigma \vdash_{\mathcal{I}} \varphi_j$ for each $j \in [1, k+h-1]$, and for each leaf node $\mathsf{N}_{\mathsf{leaf}} \in S_{i+1}$, there exists a node $\mathsf{N_j}$ = '$\mathsf{R_i} : \mathsf{t_j}$' on PATH$(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}})$ such that $\Upsilon(\mathsf{N_j}) = (R_i[C_{1_j}, \ldots, C_{m_j}; U_{p_j}], t_j[U_{p_j}])$ and $t_j[U_{p_j}] = t_{p_{\varphi_j}}[U_{p_j}]$. It suffices to show that we only need to consider those nodes $\mathsf{N_j}$ ($j \in [1, k+h-1]$) in T_i. For if this holds, then the same argument for Case 1 can verify that the claim holds on T_{i+1}. We show this by distinguishing the following cases.

(a) For each leaf node $\mathsf{N}_{\mathsf{leaf_j}}$ ($j \in [2,k]$), the node $\mathsf{N_j}$ on PATH$(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf_j}})$ must be in T_i since $\mathsf{N}_{\mathsf{leaf_j}}$ appears in T_i.

(b) For each new leaf nodes $\mathsf{N}_{\mathsf{f_j}}$ ($j \in [1,h]$) in S_{new}, there exist PATH$(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{f_j}})$ and PATH$(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}_1})$ in tree T_{i+1}, where there is an edge from $\mathsf{N}_{\mathsf{leaf}_1}$ to $\mathsf{N}_{\mathsf{f_j}}$. In this

case, the leaf node $\mathsf{N}_{\mathsf{f_j}}$ is the only node appearing on $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{f_j}})$, but not on $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}_1})$. That is, if the node $\mathsf{N_j}$ ($j \in [1,h]$) on $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{f_j}})$ is not in tree T_i, it must be the leaf node $\mathsf{N}_{\mathsf{f_j}}$ (see Fig. 2.5).

If there exists such a node $\mathsf{N_j}$ ($j \in [1,h]$) that is not in S_{new}, it must be on $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{f_j}})$ and hence, there exists $\text{PATH}(\mathsf{N_j}, \mathsf{N}_{\mathsf{f_j}})$ for each $\mathsf{N}_{\mathsf{f_j}}$ in S_{new}. In this case, we only need to consider this $\mathsf{N_j}$ in the tree T_i.

If such a node $\mathsf{N_j}$ does not exist, we show that we can use the node N instead of $\mathsf{N_j}$, where N is in T_i. In this case, for each $j \in [1,h]$, the node $\mathsf{N_j}$ must be the leaf node $\mathsf{N}_{\mathsf{f_j}}$, and the CIND φ_j must be in the form of $(R_{g'}[U'_2; U_{p_2}, U_f] \subseteq R_{b'}[D_1, \ldots, D_m; V_p], t_{p_{\varphi_j}})$ such that $t_{p_{\varphi_j}}[U_{p_2}] = t_{p_\phi}[U_{p_2}]$ and $t_{p_{\varphi_j}}[U_f] = \rho_j$. Here U_f is the list of all finite-domain attributes in $\mathtt{attr}(R_{g'}) \setminus (U' \cup U_{p_2})$, and $\rho_{U_f} = \{\rho_1, \ldots, \rho_h\}$ is the set of all possible instantiations of U_f.

We show that we can use N instead of $\mathsf{N}_{\mathsf{f_j}}$, and use a CIND φ_g derived below instead of φ_j for $j \in [1,h]$.

- Since $\Sigma \vdash_{\mathcal{I}} \varphi_j$ for each $j \in [1,h]$, we have that $\Sigma \vdash_{\mathcal{I}} \varphi_{g'}$, where $\varphi_{g'} = (R_{g'}[U'_2; U_{p_2}] \subseteq R_{b'}[D_1, \ldots, D_m; V_p], t_{p_{\varphi_{g'}}})$, and $t_{p_{\varphi_{g'}}}[U_{p_2} \parallel V_p] = t_{p_{\varphi_1}}[U_{p_2} \parallel V_p]$, by using CIND7.
- By applying CIND2 to the CIND ψ', we have that $\Sigma \vdash_{\mathcal{I}} \psi''$, where $\psi'' = (R_g[U'; U_{p_1}] \subseteq R_{g'}[U'_2; U_{p_2}], t_{p_{\psi''}})$, and $t_{p_{\psi''}} = t_{p_{\psi'}}$.
- By applying CIND5 to the CIND ψ'', we have that $\Sigma \vdash_{\mathcal{I}} \psi'''$, where $\psi''' = (R_g[U'; U'_p] \subseteq R_{g'}[U'_2; U_{p_2}], t_{p_{\psi'''}})$, $t_{p_{\psi'''}}[U'_p] = t_g[U'_p]$, and $t_{p_{\psi'''}}[U_{p_2}] = t_{p_{\psi''}}[U_{p_2}]$.
- By applying CIND3 to ψ''' and $\varphi_{g'}$, we can get that $\Sigma \vdash_{\mathcal{I}} \varphi_g$, where $\varphi_g = (R_g[U'; U'_p] \subseteq R_{b'}[D_1, \ldots, D_m; V_p], t_{p_{\varphi_g}})$, $t_{p_{\varphi_g}}[U'_p] = t_{p_{\psi'''}}[U'_p] = t_g[U'_p]$ and $t_{p_{\varphi_g}}[V_p] = t_{p_{\varphi_{g'}}}[V_p]$.

Since $\Upsilon(\mathsf{N}) = (R_g[U'; U'_p], t_g[U'_p])$, we can use N and φ_g instead of $\mathsf{N}_{\mathsf{f_j}}$ and φ_j ($j \in [1,h]$), which still satisfy the conditions in the claim. Hence we only need to use nodes in T_i, on which the claim holds by the induction hypothesis.

(5) Finally, we show that if $\Sigma \models \psi$, then $\Sigma \vdash_{\mathcal{I}} \psi$, based on Claim 2.3.5. Let T_f be a result $\mathsf{Chase}(\Sigma, \psi)$ of the chase process.

Recall that for each leaf $\mathsf{N}_{\mathsf{leaf}}$ of T_f, $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}})$ represents a database instance D of $\mathcal{R}$. Observe that $D \models \psi$. Indeed, $D \models \Sigma$ since no CINDs in Σ are applicable to any nodes in T_f. By $\Sigma \models \psi$, we have that $D \models \psi$.

These tell us that for each leaf node $\mathsf{N}_{\mathsf{leaf}}$ in T_f, there must exist a node $\mathsf{N} = \text{`}\mathsf{R_b} : \mathsf{t_b}\text{'}$ on $\text{PATH}(\mathsf{N}_{\mathsf{root}}, \mathsf{N}_{\mathsf{leaf}})$ such that $t_b[B_1, \ldots, B_m] = (v_1, \ldots, v_m)$ and $t_b[Y_p] = t_{p_\psi}[Y_p]$. Here $\Upsilon(\mathsf{N}) = (R_b(B_1, \ldots, B_m; Y'_p), t[Y'_p])$, where $Y_p \subseteq Y'_p$. Hence for each $\mathsf{N}_{\mathsf{leaf}}$, we can verify

the following using $\mathcal{I}$:

$\varphi_1 = (R_b[B_1, \ldots, B_m; Y'_p] \subseteq R_b[B_1, \ldots, B_m; Y'_p], t_{p_{\varphi_1}})$, where $t_{p_{\varphi_1}}[Y'_{pL}] = t_{p_{\varphi_1}}[Y'_{pR}] = t_b[Y'_p]$ CIND1

$\varphi_2 = (R_b[B_1, \ldots, B_m; Y'_p] \subseteq R_b[B_1, \ldots, B_m; Y_p], t_{p_{\varphi_2}})$, where $t_{p_{\varphi_2}}[Y'_p] = t_b[Y'_p]$ and $t_{p_{\varphi_2}}[Y_p] = t_b[Y_p]$ φ_1, CIND6

That is, for each $\mathsf{N}_{\mathsf{leaf}}$, $\Sigma \vdash_{\mathcal{I}} \varphi_2$. Taking this together with the existence of $\mathsf{N} = \text{`}\mathsf{R_b} : \mathsf{t_b}\text{'}$, we have that $\Sigma \vdash_{\mathcal{I}} \psi$ by Claim 2.3.5. That is, if $\Sigma \models \psi$, then $\Sigma \vdash_{\mathcal{I}} \psi$.

This completes the proof for the completeness of $\mathcal{I}$ for CINDs when finite-domain attributes may be present. □

Unrestricted implication. We have so far only considered *finite* implication, when *finite* databases are considered, *i.e.,* for database instances in which each relation has a finite set of tuples. A CIND ψ is finitely implied by Σ if for every *finite* database D, if $D \models \Sigma$, then $D \models \psi$. For theoretical interest one may also want to consider *unrestricted* implication, where ψ is implied by Σ if for every database D, either *finite* or *infinite*, if $D \models \Sigma$, then $D \models \psi$. To distinguish these, we denote *finite* implication and *unrestricted* implication by $\Sigma \models_{fin} \psi$ and $\Sigma \models_{unr} \psi$, respectively.

The result below tells us that, however, for CINDs these notions are equivalent. As a result, we can focus on finite implication for CINDs, and use $\Sigma \models \psi$ to denote both $\Sigma \models_{fin} \psi$ and $\Sigma \models_{unr} \psi$.

Proposition 2.3.6 *Finite implication and unrestricted implication coincide for* CINDs.

Proof: By definition, unrestricted implication entails finite implication. That is, if $\Sigma \models_{unr} \psi$, then $\Sigma \models_{fin} \psi$.

Conversely, it is easy to verify that the inference system $\mathcal{I}$ is sound for unrestricted implication, *i.e.,* if $\Sigma \vdash_{\mathcal{I}} \psi$, then $\Sigma \models_{unr} \psi$. Moreover, by Theorem 2.3.4, if $\Sigma \models_{fin} \psi$, then $\Sigma \vdash_{\mathcal{I}} \psi$. From these it follows that finite implication entails unrestricted implication, *i.e.,* if $\Sigma \models_{fin} \psi$, then $\Sigma \models_{unr} \psi$. □

2.3.2.2 The Complexity of the Implication Analysis

We next establish the computational complexity bounds for the implication analysis of CINDs. We investigate the problem again in two settings, namely, in the absence of finite-domain attributes and in the general setting.

In the absence of finite-domain attributes. It is known that for standard INDs in this setting, the implication problem is PSPACE-complete [CFP84]. Below we show that

in this setting, the implication problem for CINDs retains the same complexity as their standard counterpart.

Theorem 2.3.7 *The implication problem for* CINDs *is* PSPACE*-complete in the absence of attributes with finite domains.*

Proof: It is known that the implication problem for INDs is PSPACE-complete in the absence of finite-domain attributes [CFP84]. Since CINDs subsume INDs, the implication problem for CINDs is also PSPACE-hard.

We next show that the implication problem for CINDs is in PSPACE in the absence of finite-domain attributes. We show this by giving a linear space non-deterministic algorithm for deciding whether $\Sigma \models \psi$, along the same lines as its counterpart for INDs (see [AHV95, CFP84]). If this holds, then by Savitch's theorem [Sav70], there is a deterministic quadratic-space algorithm for checking whether $\Sigma \models \psi$, and therefore, the implication problem is in PSPACE.

Indeed, the chase procedure developed in the proof of Theorem 2.3.2 gives such an algorithm. Consider a set $\Sigma \cup \{\psi\}$ of CINDs over a database schema $\mathcal{R} = (R_1, \ldots, R_n)$, where $\psi = (R_a[A_1, \ldots, A_m; X_p] \subseteq R_b[B_1, \ldots, B_m; Y_p], t_{p_\psi})$. Recall that the chase process starts with an initial database D_0, which contains a single tuple $t_a \in I_a$ such that $t_a[A_i] = v_i$ for all $i \in [1, m]$ and $t_a[X_p] = t_{p_\psi}[X_p]$. As we have seen in the proof of Theorem 2.3.2, $\mathsf{Chase}(\Sigma, \psi) \models \psi$, where $\mathsf{Chase}(\Sigma, \psi)$ is the final database generated by the chase process. Moreover, if $\Sigma \models \psi$, then there must exist a tuple $t_b \in I_b$ in $\mathsf{Chase}(\Sigma, \psi)$ such that $t_b[B_i] = v_i$ for $i \in [1, m]$, and $t_b[Y_p] = t_{p_\psi}[Y_p]$. More specifically, there exists a finite sequence $< t_0, \ldots, t_l >$ of tuples such that $t_0 = t_a$, $t_l = t_b$, and for each $i \in [0, l-1]$, t_{i+1} is obtained by applying a CIND ψ' in Σ to t_i.

Based on these, a linear space non-deterministic algorithm can be developed as follows:

- Initialize a single tuple $t_0 = t_a \in I_a$.
- Replace tuple t_i with t_{i+1} if t_{i+1} can be derived from t_i by applying a CIND ψ' in Σ to t_i by using CIND1–CIND6. There are possibly multiple such t_{i+1}'s. The algorithm non-deterministically picks one of them.
- Repeat these steps until no more changes can be made.
- If tuple $t_b \in I_b$, return 'yes'; and return 'no' otherwise.

As shown in the proof of Theorem 2.3.2, if $\Sigma \models \psi$, then t_b is in $\mathsf{Chase}(\Sigma, \psi)$ and hence, the algorithm returns 'yes'. Conversely, if the algorithm returns 'yes', *i.e.,* when t_b is in $\mathsf{Chase}(\Sigma, \psi)$, then by Claim 2.3.3, $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$. By Theorem 2.3.2, $\Sigma \models \psi$.

Hence the algorithm correctly determines whether $\Sigma \models \psi$. As a result, the implication problem for CINDs is in PSPACE in the absence of finite-domain attributes. □

In the general setting. When finite-domain attributes are present, the implication analysis of CINDs becomes more involved. Nevertheless, the increased complexity is not incurred only by the presence of finite-domain attributes. Indeed, a close examination of the proofs of [AHV95, CFP84] reveals that while the PSPACE-completeness for the implication analysis of standard INDs was established for relations with infinite domains only, the proofs remain intact when finite-domain attributes are present. That is, the following result holds.

Corollary 2.3.8 *In the presence of finite-domain attributes, the implication problem for* INDs *remains* PSPACE*-complete.*

Theorem 2.3.7 and Corollary 2.3.8 together tell us that neither finite-domain attributes nor constant patterns alone complicate the implication problem. However, as we shall show in Theorem 2.3.10, when they are taken together, the implication analysis becomes more intriguing. That is, their interaction makes the implication problem for CINDs harder.

Before we prove Theorem 2.3.10, we first examine the chase process introduced in the proof of Theorem 2.3.4. Given a set $\Sigma \cup \{\psi\}$ of CINDs, the chase procedure inspects whether $\Sigma \models \psi$. Below we give its computational complexity.

Proposition 2.3.9 *Given a set $\Sigma \cup \{\psi\}$ of* CINDs *defined on a database schema $\mathcal{R}$, the chase procedure given in the proof of Theorem 2.3.4 terminates in $O(2^{2^{n^2}})$ time, where n is the size of the input,* i.e., *the size of $\mathcal{R}$, Σ and ψ.*

Proof: The chase procedure is obviously in $O(|T_f|)$ time, where T_f is a result $\mathsf{Chase}(\Sigma, \psi)$ of the chase process, and $|T_f|$ is the number of nodes in T_f. Recall that the each root-to-leaf path of T_f represents a database of schema $\mathcal{R}$. Hence the depth of T_f is bounded by the maximum size $|I|$ of a database instance of $\mathcal{R}$. Moreover, the maximum number of children of a node in T_f is also bounded by $|I|$. We show that $|I|$ is in $O(2^{n^2})$ as follows.

- The cardinality of a finite domain in $\mathcal{R}$ is a constant specified in the schema $\mathcal{R}$, *i.e.,* it is bounded by n.
- For an infinite domain in $\mathcal{R}$, the chase process uses only those constants appearing in the patterns in Σ or ψ, and the finite set $\{v_0, \ldots, v_m\}$ of variables (bounded by the size of ψ). These are also bounded by the input size n.

Therefore, $|I|$ is bounded by $O(n^n) = O(2^{log(n)*n}) \leq O(2^{n^2})$. Hence $|T_f|$ is in $O((2^{n^2})^{2^{n^2}}) = O(2^{2^{n^2}})$. □

We are now ready to give the complexity bound for the implication analysis of CINDs in the general setting.

Theorem 2.3.10: *In the general setting, the implication problem for* CINDs *is* EXPTIME*-complete.* □

Proof: (1) We first show that the problem is in EXPTIME. Given a set $\Sigma \cup \{\psi\}$ of CINDs on a relational schema $\mathcal{R}$, we develop an algorithm in $O(2^{n^k})$ time, where k is a constant and n is the size of $\mathcal{R}$, Σ and ψ, such that it returns 'yes' if and only if $\Sigma \models \psi$. Assume that $\mathcal{R} = (R_1, \ldots, R_n)$, and that $\psi = (R_a[A_1, \ldots, A_m; X_p] \subseteq R_b[B_1, \ldots, B_m; Y_p], t_{p_\psi})$.

Proposition 2.3.9 tells us that the *chase* procedure given in the proof of Theorem 2.3.4 cannot be used as such an algorithm, since it is doubly exponential. Nevertheless, we shall develop a singly exponential-time algorithm based on the chase procedure. Indeed, the complexity of the chase process is incurred by redundant nodes in the trees generated, as shown in Fig. 2.4. However, we can use graphs instead of trees to remove the redundancy.

Observe the following. Every node in a tree T is reachable from the root node '$R_a : t_a$'. In addition, if $\Sigma \models \psi$, then from each node in T there exists a path to a node N = '$R_b : t_b$' such that $t_b[B_1, \ldots, B_m] = t_a[A_1, \ldots, A_m]$ and $t_b[Y_p] = t_{p_\psi}[Y_p]$. One can check whether there exists a path from a node to another in quadratic time [Pap94].

We now develop an EXPTIME algorithm based the chase procedure. The main idea is to maintain a directed edge-labeled graph $G(V, E, L, \Sigma)$ and a mapping $H(V)$. Here (a) the V consists of nodes in the form of '$\mathsf{R_i : t_i}$', where R_i is a relational schema in $\mathcal{R}$, and t_i is an R_i tuple taking only values from the active domains defined before; (b) the Σ is the set of CINDs, and (c) the L is a relation defined on E such that for each $e \in E$, $L(e) \subseteq \Sigma$. Given a node u in V of G, $H(\mathsf{u})$ is the set of CINDs in Σ that are already applied to the node u in the process. With these two data structures, we can avoid *unnecessary* computations.

Below we first present the algorithm, and then verify the correctness of the algorithm. Finally, we show that the algorithm is in exponential time.

We first present the algorithm.

(a) It initializes the Σ to be the set of CINDs, the node set $V = \{\mathsf{N_{root}}\}$, the edge set $E = \emptyset$, and $H(\mathsf{N_{root}}) = \emptyset$. Here the node $\mathsf{N_{root}}$ is the root node '$\mathsf{R_a : t_a}$' of a tree T in the chase process.

(b) For each node u = '$\mathsf{R_i} : \mathsf{t_i}$' in V, it checks whether there exists a CIND $\psi' = (R_i[U;U_p] \subseteq R_j[V;V_p], t_{p_{\psi'}})$ in Σ, but not in $H(\mathsf{u})$, such that $t_i[U_p] = t_{p_{\psi'}}[U_p]$.
(c) If there exists such a CIND^p ψ' for the node u = '$\mathsf{R_i} : \mathsf{t_i}$', it first generates a set S_{new} of new nodes, and then updates the graph G and the mapping H accordingly.

The set S_{new} is generated along the same lines as the chase process in the proof of Theorem 2.3.4.

- Let V_f be the list of all finite-domain attributes in $\mathsf{attr}(R_j) \setminus (V \cup V_p)$; and let $\rho(V_f)$ be an instantiation of V_f. That is, for each attribute $C \in V_f$, $\rho(C)$ is a concrete data value, drawn from the finite domain $\mathsf{dom}(C)$.
- For each possible instantiation $\rho(V_f)$, it generates a *new* node u'_ρ = '$\mathsf{R_j} : \mathsf{t}_\rho$' such that $t_\rho[V] = t_i[U]$, $t_\rho[V_p] = t_{p_{\psi'}}[V_p]$, $t_\rho[V_f] = \rho(V_f)$, and $t_\rho[C] = v_0$ for all the other attributes C in $\mathsf{attr}(R_j)$.

Then, for the mapping H, $H(\mathsf{u}) = H(\mathsf{u}) \cup \{\psi'\}$, and for each node u' in $S_{new} \setminus V$, $H(\mathsf{u}) = \emptyset$. For the graph $G(V,E,L,\Sigma)$,

- its edge set E is updated as follows: for each node u' in S_{new}, it adds an edge $e = (\mathsf{u},\mathsf{u}')$ to E, and sets $L(e) = \{\psi'\}$ if u' is not in V, or $L(e) = L(e) \cup \{\psi'\}$, otherwise; and
- its node set V is updated to be $V \cup S_{new}$.

(d) The above process repeats until there are no more changes for the node set V and the edge set E of the graph $G(\Sigma,V,E)$. We denote the final resulting graph as G_f.
(e) The algorithm finally checks whether $\Sigma \not\models \psi$ based on the graph G_f.

- Let S_b be the set of nodes '$R_b : t_b$' in G_f such that $t_b[B_1,\ldots,B_m] = (v_1,\ldots,v_m)$ and $t_b[Y_p] = t_{p_\psi}[Y_p]$.
- It then recursively enlarges the set S_b to include those nodes u in V such that all neighboring nodes u' of u with edges $e = (\mathsf{u},\mathsf{u}')$ labeled with ψ', *i.e.,* $\psi' \in L(e)$, are already included in S_b. Here ψ' is any CIND in Σ.
- After the set S_b reaches a fixpoint, if the node $\mathsf{N_{root}}$ = '$\mathsf{R_a} : \mathsf{t_a}$' is in S_b, the algorithm simply returns 'yes', and returns 'no' otherwise.

We next show that $\Sigma \models \psi$ iff the algorithm returns 'yes'. Indeed, the algorithm simulates the chase procedure given in the proof of Theorem 2.3.4. If it returns 'no', one can readily expand G_f into a tree, which represents a database instance D (see the proof of Theorem 2.3.4) such that $D \models \Sigma$, but not $D \models \psi$, *i.e.,* $\Sigma \not\models \psi$. If it returns 'yes', then $\Sigma \models \psi$ by Claim 2.3.5 given in the proof of Theorem 2.3.4.

To see that the algorithm is in exponential time, observe the following. (a) The number of nodes in the graph G_f is bounded by $|I|$, where$|I|$ is the maximum size

of a database instance. Therefore, the size of G_f is bounded by $O(2^{n^2})$ as argued in the proof of Proposition 2.3.9. (b) For the set Σ of CINDs, the number of equivalent CINDs in the special form is bounded by $|I| = O(2^{n^2})$ as indicated in the proof of Theorem 2.3.4, and the number of CINDs equivalent to ψ in the special form is also bounded by $I = O(2^{n^2})$. (c) The size of S_b is bounded by the number of nodes in G_f, *i.e.,* in $O(2^{n^2})$. From these it follows that graph G_f can be constructed in $O(2^{n^2} * 2^{n^2})$ $= O(2^{n^2})$ time, and constructing the set S_b can be done in $O(2^{n^2} * 2^{n^2} * 2^{n^2}) = O(2^{n^2})$ time. Based on these one can readily verify that the algorithm is indeed in EXPTIME.

(2) We next show that the problem is EXPTIME-hard, by reduction from the two-player game of corridor tiling problem (TPG-CT), which is EXPTIME-complete [Chl86, vEB97].

An instance of TPG-CT consists of a tiling system $(X, H, V, \vec{t}, \vec{b})$ and a positive integer n, where X is a finite set of tiles (dominoes), $H, V \subseteq X \times X$ are two binary relations, $\vec{t}$ and $\vec{b}$ are two n-vectors of given tiles in X, and n is the number of columns (the width of the corridor). It is to determine whether or not player I has a winning strategy for tiling the corridor. By tiling the corridor we mean that there exists a tiling $\tau : \mathbb{N} \times \mathbb{N} \to X$ and a positive integer m such that for all $x \in [1, n]$ and $y \in [1, m]$, the *tiling adjacency conditions* are observed, *i.e.,*

- if $\tau(x, y) = d$ and $\tau(x+1, y) = d'$, then $(d, d') \in H$, *i.e.,* horizontally adjacent tiles have matching "colors";
- if $\tau(x, y) = d$ and $\tau(x, y+1) = d'$, then $(d, d') \in V$, *i.e.,* vertically adjacent tiles have matching colors; and
- $\tau(x, 1) = \vec{t}[x]$ and $\tau(x, m) = \vec{b}[x]$, where $\vec{t}[x]$ (resp. $\vec{b}[x]$) denotes the x-th element of the vector $\vec{t}$ (resp. $\vec{b}$); that is, the given tiles of $\vec{t}$ and $\vec{b}$ are placed on the top and the bottom rows, respectively. The given tiles $\vec{t}$ are placed on the top row by the referee of the game.

Each player in turn places a tile from X in the first free location (column by column from left to right, and row by row from top to bottom), observing the tiling adjacency conditions. Player I wins if either Player II makes an illegal move by placing a tile that violates one of the adjacent conditions, or if the bottom row $\vec{b}$ is placed. Player I has a winning strategy iff Player I can always win no matter how Player II plays. The problem is already EXPTIME-complete when n is *odd* [Chl86, vEB97], and thus below we assume that n is an *odd* number, and that Player I makes the first move.

Given an instance of TPG-CT $(X, H, V, \vec{t}, \vec{b})$ and n, we define a relational schema

I_R:

K	A_0	A_1	A_2	A_3	A_4	next	P	Z
k_1	a_1	a_2	a_3	a_4	b	k_2	1	#
k_2	a_2	a_3	a_4	b	c	k_3	2	#
k_3	a_3	a_4	b	c	d	k_4	3	#
k_4	a_4	b	c	d	e	k_5	4	#
k_5	b	c	d	e	f	k_6	1	#
...								
...								
k_x	x	b_1	b_2	b_3	b_4	k_1	4	#

I_S:

B
b

Figure 2.6: Encoding of a TPG-GT instance with n = 4, $\vec{t}$ = (a_1,a_2,a_3,a_4) and $\vec{b}$ = (b_1,b_2,b_3,b_4) for the proof of Theorem 2.3.10

$\mathcal{R}$, a set of CINDs Σ and a CIND ψ such that $\Sigma \not\models \psi$ if and only if there is a winning strategy for Player I. If this holds, then the problem is EXPTIME-hard. Indeed, this problem is the complement problem of the implication problem, from which it follows that the implication problem for CINDs is also EXPTIME-hard.
(a) The database schema $\mathcal{R}$ consists of two relation schemas $R(K, A_0, A_1, \ldots, A_n,$ next, P, $Z)$ and $S(B)$, where for all $i \in [0,n]$, A_i has a finite domain $\mathsf{dom}(A_i) = X$, P has a finite domain $\mathsf{dom}(P) = \{1,\cdots,n\}$, the domains of attributes K and next are positive integers, and Z has a finite domain with two symbols # and !. The attribute B has a finite domain consisting of two distinct values: c and b.

Intuitively, an R tuple t encodes a placement of tiles in the game. More specifically, tuple t is the snapshot of the game showing the last $n+1$ plays, where a) $t[A_n]$ is the *new* tile placed by a player, b) $t[A_0,\ldots,A_{n-1}]$ consists of the n tiles placed before $t[A_n]$, c) $t[P]$ codes the horizontal position of tile $t[A_n]$ in a row, d) $t[K]$ and $t[\mathsf{next}]$ encode a list of such snapshots: $t[K]$ is the "identifier" of the current snapshot, while $t[\mathsf{next}]$ is a pointer to the next one (See Fig. 2.6). In addition, $t[Z]$ indicates that the game continues when it is #, and that the game should stop when it is !. If Player I makes an illegal move, we indicate it with the presence of a tuple s of schema S with $s[B]$ = 'c'.

We want to show that if there exists an instance $D = (I_R, I_S)$ of $\mathcal{R}$ such that D satisfies Σ, but not ψ, if and only if Player I has a winning strategy.

(b) We next define the set Σ of CINDs that encodes the play. We use N_{odd} and N_{even} to denote the set of *even* numbers and the set of *odd* numbers in $[1,n]$, indicating the moves of Player I and Player II, respectively.

Initial condition. The top row of the corridor has to be set to $\vec{t}$. We use a CIND φ_1 to ensure that if I_S is nonempty, then there exists an R tuple t such that $t[A_0,\ldots,A_{n-1}]$ matches $\vec{t}$.

$$\varphi_1 = (S[\mathsf{nil};\mathsf{nil}] \subseteq R[\mathsf{nil};A_0,\ldots,A_{n-1},P,Z],\ t_{p_{\varphi_1}}),$$
$$\text{where } t_{p_{\varphi_1}}[A_0,\ldots,A_{n-1}] = \vec{t} \text{ and } t_{p_{\varphi_1}}[P,Z] = (1,\#);$$

Adjacency constraints. The vertical and horizontal tiling conditions must hold. For each R tuple t, $t[A_n]$ corresponds to the tile directly under tile $t[A_0]$, and $t[A_n]$ is the tile placed next to $t[A_{n-1}]$ in a row. Thus for each tuple $t \in I_R$, we must ensure that $(t[A_0],t[A_n]) \in V$ and that $(t[A_{n-1}],t[A_n]) \in H$.

The adjacency constraints are enforced by two sets Σ_V and Σ_H of CINDs below. These CINDs assert the following: a) if Player I makes an illegal move, then I_S contains a tuple ('c'); and b) if any Player makes an illegal move, then the game should stop, by adding an R tuple t' with $t'[Z] =$!.

Σ_V: $(R[\mathsf{nil};A_0,A_n,P] \subseteq S[\mathsf{nil};B],\ t^v_{(x,y)})$,
where $t^v_{(x,y)}[A_0,A_n,h \parallel B] = (x,y,h \parallel c)$ for all $(x,y) \in (X \times X) \setminus V$ and all $h \in N_{odd}$
$(R[\mathsf{next},A_1,\ldots,A_{n-1};A_0,A_n,P] \subseteq R[K,A_0,\ldots,A_{n-2};P,A_{n-1},Z],\ t^v_{(x,y)})$,
where $t^v_{(x,y)} = (x,y,h \parallel h+1,y,!)$ for all $h \in [1,n-1]$ and all $(x,y) \in (X \times X) \setminus V$

Σ_H: $(R[\mathsf{nil};A_{n-1},A_n,P] \subseteq S[\mathsf{nil};B],\ t^h_{(x,y)})$,
where $t^h_{(x,y)} = (x,y,h \parallel c)$ for all $(x,y) \in (X \times X) \setminus H$ and all $h \in N_{odd}$
$(R[\mathsf{next},A_1,\ldots,A_{n-2};A_{n-1},A_n,P] \subseteq R[K,A_0,\ldots,A_{n-3};P,A_{n-2},A_{n-1},Z],\ t^v_{(x,y)})$,
where $t^v_{(x,y)} = (x,y,h \parallel h+1,x,y,!)$ for all $h \in [1,n-1]$ and all $(x,y) \in (X \times X) \setminus H$.

Player I has to respond to all possible legal moves of Player II. For each R tuple t_1 that is a legal move, if $t_1[P]$ is even, *i.e.,* if the last move $t_1[A_n]$ was made by Player II, then for each tile $x \in X$ that satisfies the horizontal constraint $(t_1[A_n],x) \in H$ and the vertical constraint $(t_1[A_0],x) \in V$, there must exist a tuple t_2 in I_R with $t_2[K] = t_1[\mathsf{next}]$, $t_2[A_{n-1}] = x$ and moreover, $t_2[A_i] = t_1[A_{i+1}]$ for $i \in [0,n-1]$. That is, all possible *legal* moves of Player II have to be considered. We encode this with a set $\Sigma_\forall$ of CINDs.

$\Sigma_\forall$: $(R[\mathsf{next},A_1,A_2,\ldots,A_{n-2};\ P,A_0,A_{n-1},A_n,Z] \subseteq R[K,A_0,A_1,\ldots,A_{n-2};\ P,A_{n-1},Z],\ t_{(x,y,w)})$,
where $t_{(x,y,w)} = (h,x,y,w,\# \parallel h+1,w,\#)$, for all $h \in N_{even}$, and for all $(x,w) \in V$ and $(y,w) \in H$

Play continues unless Player I has won. For each R tuple t_1, if $t_1[P] < n$ and $t_1[Z] =$ #, then there must exist some tuple t_2 such that $t_2[K] = t_1[\mathsf{next}]$, $t_2[A_0,\ldots,A_{n-1}] = t_1[A_1,\ldots,A_n]$ and $t_2[P] = t_1[P]+1$. If $t_1[P] = n$ and if the bottom vector $\vec{b}$ is not matched, *i.e.,* if for some $i \in [1,n]$, $t_1[A_i] \neq \vec{b}[i]$, then there must exist some t_2 such that $t_2[K] = t_1[\mathsf{next}]$, $t_2[A_0,\ldots,A_{n-1}] = t_1[A_1,\ldots,A_n]$ and $t_2[P] = 1$. We express this as a set Σ_p consisting of the following CINDs:

$$\varphi_h = (R[\mathsf{next},A_1,\ldots,A_n;\ P,Z] \subseteq R[K,A_0,\ldots,A_{n-1};\ P,Z], t_{p_h}),$$
$$\text{where for each } h \in [1,n-1],\ t_{p_h} = (h,\# \parallel h+1,\#).$$
$$\varphi_{(i,x)} = (R[\mathsf{next},A_1,\ldots,A_i,A_{i+2},\ldots,A_n;\ P,A_{i+1},Z]$$
$$\subseteq R[K,A_0,\ldots,A_{i-1},A_{i+1},\ldots,A_{n-1};\ P,A_i,Z],\ t^n_{i,x}),$$
$$\text{where } t^n_{(i,x)} = (n,x,\# \parallel 1,x,\#), \text{ for all } i \in [1,n] \text{ and all } x \in X \setminus \{\vec{b}[i]\}.$$

Observe that if an illegal move was made, a next move t_2 is added with $t_2[Z] =!$ and $t_2[P] = h+1$, by Σ_H or Σ_V.

The set Σ consists of all the CINDs given above, *i.e.,* $\Sigma = \{\varphi_1\} \cup \Sigma_V \cup \Sigma_H \cup \Sigma_p \cup \Sigma_\forall$. It is easy to see that The number of CINDs in Σ is bounded by a polynomial of n and the number of tiles in X.

(c) We define CIND $\psi = (S[\mathsf{nil};\mathsf{nil}] \subseteq S[\mathsf{nil};B], (\mathsf{nil} \parallel c))$.

Intuitively, if $D \not\models \psi$ then a) I_S is nonempty, and b) there exists no tuple $t \in I_S$ such that $t[B] =$ 'c', *i.e.,* Player I does not make illegal move.

The reduction is obviously in polynomial time. We next verify that Player I has a winning strategy iff $\Sigma \not\models \psi$.

First, suppose that $\Sigma \not\models \psi$. Then there exists an instance $D = (I_R, I_S)$ of $\mathcal{R}$ such that $D \models \Sigma$, but $D \not\models \psi$. We give a wining strategy for Player I. Player I begins with the tuple in R enforced by φ_1 in Σ. Such a tuple must exist because I_S must be nonempty (by $D \not\models \psi$) and hence, φ_1 is applicable. At any step in the game, there is a tuple in I_R that represents the last $n+1$ moves of the play thus far. For each valid tile x_j that Player II places as the next move, represented by t, the CINDs in $\Sigma_\forall$ ensure the existence of a tuple t' with $t[\mathsf{next}] = t'[K]$ and $t'[A_{n-1}] = x_j$ and $t'[A_i] = t[A_{i+1}]$ for $i \in [0,n-1]$, *i.e.,* a response from Player I. By $D \not\models \psi$ and $D \models \Sigma_H \cup \Sigma_V \cup \Sigma_\forall$, the move t' satisfies both the horizontal and the vertical constraints, and it also correctly updates the last $n+1$ tiles played. Furthermore, by $D \models \Sigma_p$, the play continues until Player I wins. Thus Player I has a winning strategy.

Conversely, suppose that Player I has a winning strategy. We then form an instance $D = (I_R, I_S)$ of $\mathcal{R}$ such that I_R consists of all valid plays in any game, where each tuple

codes the horizontal position of the last move in a row and the last $n+1$ tiles played in the game, and I_S has a single tuple t such that $t[B] =$'b' (*i.e.,* Player I makes no illegal move). It is easy to confirm that $D \models \Sigma$, but $D \not\models \psi$. □

2.4 Acyclic Conditional Inclusion Dependencies

Corollary 2.3.8 and Theorem 2.3.10 tell us that it is rather expensive to reason about INDs or CINDs. Indeed, the implication analysis of INDs and CINDs are beyond reach in practice.

However, not all is lost: in practice one often needs only a fragment of INDs or CINDs with a lower complexity. That is, we do not have to pay the price of the complexity of full-fledged INDs or CINDs when we only need certain special cases of the dependencies. In the next two sections we focus on such special cases of INDs and CINDs.

One of the well-studied fragments of INDs is identified as sets of acyclic INDs [Sci86] (see *e.g.,* [AHV95]).

A set Σ of INDs over a relational schema $\mathcal{R}$ is *acyclic* if there exists no sequence $R_i[X_i] \subseteq S_i[Y_i]$ ($i \in [1,n]$) of INDs in Σ, where $R_{i+1} = S_i$ for each $i \in [1,n-1]$, and $R_1 = S_n$. We refer to such a set of INDs as a set of AINDs.

Along the same lines, we define acyclic CINDs.

Acyclic CINDs. A set Σ of CINDs is *acyclic* if there exists no sequence $(R_i[X_i;X_{pi}] \subseteq S_i[Y_i;Y_{pi}],t_{pi})$ ($i \in [1,n]$) of CINDs in Σ such that $R_{i+1} = S_i$ for $i \in [1,n-1]$, and $R_1 = S_n$. We refer to such a set of CINDs as a set of ACINDs.

It is common to find a set of CINDs acyclic in practice. For example, the set of CINDs in Example 2.2.1 and the set of CINDs in Example 2.3.3 are both acyclic.

We next show that ACINDs indeed make our lives easier, *i.e.,* they allow more efficient static analysis. By Theorem 2.3.1, any set of ACINDs is satisfiable. Hence we shall focus on the implication analysis of ACINDs, *i.e.,* the problem for determining, given a set Σ of ACINDs and a CIND φ, whether $\Sigma \models \varphi$. Note that while Σ is acyclic, $\Sigma \cup \{\varphi\}$ may be cyclic.

We study the implication problem for ACINDs in the absence of finite-domain attributes and in the general setting.

In the absence of finite-domain attributes. It has been shown that in this setting, the implication problem for AINDs is NP-complete [CK84]. We next show that the im-

plication problem for ACINDs retains the same complexity as its standard counterpart, *i.e.,* ACINDs do not complicate the implication analysis in the absence of finite-domain attributes.

Corollary 2.4.1 *The implication problem for* ACINDs *is* NP*-complete in the absence of attributes with a finite domain.*

Proof: We first show that the problem is NP-hard. The implication problem for acyclic INDs is already NP-hard [CK84]. Since acyclic INDs are a special case of acyclic CINDs, the implication problem for ACINDs is also NP-hard.

We next show that the problem is in NP.

Consider the non-deterministic algorithm given in the proof of Theorem 2.3.7 for determining whether a set Σ of CINDs implies another CIND ψ. Observe that when Σ is a set of ACINDs without finite-domain attributes, the initial tuple t_0 can be replaced by other tuples for at most n times, where n is the number of relations in the database schema $\mathcal{R}$. That is, the linear space non-deterministic algorithm runs in polynomial time. Hence, it is indeed an NP algorithm for deciding whether $\Sigma \models \psi$, and as a result, the implication problem for ACIND is in NP without finite-domain attributes. □

In the general setting. The presence of finite-domain attributes does not make the implication analysis of AINDs harder. Indeed, the NP algorithm given in [CK84] for checking the implication of AINDs still works in the presence of finite-domain attributes. Hence we have the following.

Corollary 2.4.2 *The implication problem for* AINDs *remains* NP*-complete in the general setting, when finite-domain attributes may be present.*

In contrast, in the general setting the implication problem for *acyclic* CINDs is more involved.

Theorem 2.4.3 *In the general setting, the implication problem for* ACINDs *is* PSPACE*-complete.*

Proof: (1) We first show that the implication problem for ACINDs is in PSPACE in the general setting.

We show this by giving a linear space non-deterministic algorithm for determining whether $\Sigma \not\models \psi$, *i.e.,* the complement of $\Sigma \models \psi$. This suffices. For if it holds, then (a) by Immerman–Szelepcsényi theorem [Imm88, Sze87], there exists a linear space

non-deterministic algorithm for determining whether $\Sigma \models \psi$; and (b) by Savitch's theorem [Sav70], there is a deterministic quadratic-space algorithm for checking whether $\Sigma \models \psi$. Therefore, the problem is in PSPACE.

Consider a set $\Sigma \cup \{\psi\}$ of acyclic CINDs defined over a database schema $\mathcal{R} = (R_1, \ldots, R_n)$. Assume *w.l.o.g.* that for any two schemas R_i and R_j ($i, j \in [1,n]$ and $j < i$), there exist no CINDs in the form of $(R_i[U;U_p] \subseteq R_j[V;V_p], t_p)$ in Σ. Observe that we can always rearrange the schemas in $\mathcal{R}$ to satisfy this condition since the CINDs in Σ are acyclic. We also assume *w.l.o.g.* that the CIND ψ is $(R_1[A_1,\ldots,A_m;X_p] \subseteq R_n[B_1,\ldots,B_m;Y_p], t_{p_\psi})$; the proof is similar for the cases where ψ is from R_i to R_j when $i \neq 1$ or $j \neq n$.

The proof consists of three parts. (a) We first introduce notations to be used in the algorithm. (b) We then present the algorithm. (c) Finally, we show that the algorithm is correct and that it is in PSPACE.

(1.1) Before we present the algorithm, we first introduce the following notations to be used in the algorithm.
(a) Let $\{\Sigma_1,\ldots,\Sigma_{n-1}\}$ be the partition of Σ such that $\bigcup_{i=1}^{n-1} \Sigma_i = \Sigma$, and for each $i \in [1,n-1]$, Σ_i is the set of CINDs of the form $(R_i[U;U_p] \subseteq R_j[V;V_p], t_p)$ such that $j \in [i+1,n]$.
(b) The number of CINDs in Σ_i ($i \in [1,n-1]$) is denoted as n_i. We assume *w.l.o.g.* that for each $i \in [1,n-1]$, $n_i > 0$, *i.e.,* there exists at least one CIND in each Σ_i.
(c) All CINDs in Σ_i ($i \in [1,n-1]$) are sorted in a certain order. We use a pointer p_i to indicate the p_i-th CIND in Σ_i. It is obvious that $1 \leq p_i \leq n_i$ for each $i \in [1,n-1]$.
(d) Given the list $P = [p_1,\ldots,p_{n-1}]$ of pointers, let $\Sigma_{P,\psi} = \{\varphi_1,\ldots,\varphi_{n-1}\}$ be the set of CINDs such that for each $i \in [1,n-1]$, φ_i is the p_i-th CIND in Σ_i.

(1.2) We now present the linear space non-deterministic algorithm for determining whether $\Sigma \not\models \psi$.

(a) It *guesses* an instantiation ρ_1 for the list X_f of all finite-domain attributes in $\mathsf{attr}(R_1) \setminus (X_p)$, where for each attribute C in X_f, $\rho_1(C)$ is a value drawn from the finite $\mathsf{dom}(C)$.

(b) It initializes a database instance $D = (I_1,\ldots,I_n)$, such that for each $i \in [1,n]$, I_i is an empty instance of schema R_i, except that I_1 contains a single tuple t_a of schema R_1, where $t_1[A_1,\ldots,A_m] = (v_1,\ldots,v_m)$, $t_1[X_p] = t_{p_\psi}[X_p]$, and $t_1[X_f] = \rho_1[X_f]$ for the list X_f of all finite-domain attributes in $\mathsf{attr}(R_1) \setminus (X_p)$. Here $v_1,\ldots,v_m$ are m distinct variables. Intuitively, t_1 encodes the LHS of the CIND ψ, and is to serve as a "witness"

for $D \not\models \psi$. We assume *w.l.o.g.* that $X_f \cap \{A_1, \ldots, A_m\}$ is *empty* since if not, in the process to be seen shortly, we can simply replace the variable with the constant $\rho[A]$ for each attribute A in $X_f \cap \{A_1, \ldots, A_m\}$.

The algorithm will ensure that for each $i \in [1, n]$, the instance I_i contains at most one tuple. This guarantees that the algorithm uses only linear space.

(c) Starting with $p_i = 1$ for all pointers p_i in P ($i \in [1, n-1]$), the algorithm processes CINDs in $\Sigma_{P,\psi}$ one by one, as follows. Let $i = 1$, and ϕ be the p_i-th CIND $(R_i[U; U_p] \subseteq R_j[V; V_p], t_{p_\phi})$ in Σ_i.

- It *guesses* an instantiation ρ_j for the list V_f of all finite-domain attributes in $\mathsf{attr}(R_j) \setminus (V \cup V_p)$, where for each C in V_f, $\rho_j(C)$ is a value drawn from the finite $\mathsf{dom}(C)$.
- If there is a tuple t_i in I_i, but there exists no tuple t_j in I_j such that $t_j[V] = t_i[U]$, $t_j[V_p] = t_{p_\phi}[V_p]$, it first creates a new tuple t'_j such that $t'_j[V] = t_i[U]$, $t'_j[V_p] = t_{p_\phi}[V_p]$, and $t'_j[V_f] = \rho_j(V_f)$, and then updates the instance I_j to contain this new tuple, *i.e.,* $I_j = \{t'_j\}$.
- If $i < (n-1)$, it increases the variable i by 1, and repeats the process above.
- Otherwise, it checks whether there exists a tuple t_n in I_n such that $t_n[B_1, \ldots, B_m] = t_1[A_1, \ldots, A_m] = (v_1, \ldots, v_m)$ and $t_n[Y_p] = t_{p_\psi}[Y_p]$.
  Observe that the current database instance $D_P \models \Sigma_{P,\psi}$.

(d) If there exists a pointer p_j ($1 \leq j \leq n-1$) such that $p_j \neq n_j$, the algorithm then adjusts the list $P = [p_1, \ldots, p_{n-1}]$ of pointers by the following pseudo-codes.

- let $j := n-1$, and increase p_{n-1} by 1;
- while ($j > 1$) do
- if $p_j = (n_j + 1)$ then
- let $p_j := 1$, and increase p_{j-1} by 1;
- decrease the variable j by 1.

(e) If there exist no such pointers in P, *i.e.,* $p_j = n_j$ for all $j \in [1, n-1]$, the algorithm stops and returns 'yes' if the tuple t_n is not found in all the cases of the list P of pointers, *i.e.,* from $[1, \ldots, 1]$ to $[n_1, \ldots, n_{n-1}]$. Otherwise, the algorithm starts again from step (a).

(1.3) We next show that the algorithm is in PSPACE and that it is correct.

Observe that at any step of the process, the database instance D contains at most n tuples, where n is the number of relation schemas in $\mathcal{R}$. Hence the non-deterministic algorithm runs in linear space.

To show the correctness of the algorithm, first observe the following.

(a) The algorithm examines each combination of those CINDs in all Σ_i's ($i \in [1, n-1]$), represented by the pointer list $P = [p_1, \ldots, p_{n-1}]$ of pointers. Recall that for each $i \in [1, n-1]$, the pointer p_i denotes the p_i-th CIND in Σ_i. For acyclic Σ, this suffices for determining whether $\Sigma \models \psi$.
(b) The algorithm examines various instantiation of variables for finite-domain attributes, by backtracking. More specifically, it changes the list $P = [p_1, \ldots, p_{n-1}]$ of pointers starting from the last pointer p_{n-1}, and does not change p_i until $p_j \geq n_j$ for each $j > i$ (recall that for each $i \in [1, n-1]$, n_i is the number of CINDs in Σ_i). This allows us to avoid random valuations of finite-domain attributes and moreover, to use the same space in the entire process.

Having seen these, we finally show the correctness of the algorithm, *i.e.,* it returns 'yes' if and only if $\Sigma \not\models \psi$.

First assume that $\Sigma \not\models \psi$. Then there exists a database instance $D = (I_1, \ldots, I_n)$ of $\mathcal{R}$ such that $D \models \Sigma$ but $D \not\models \psi$. By the definition of ψ, there must exist a tuple t_1 in the instance I_1 of schema R_1 such that $t_1[X_p] = t_{p_\psi}[X_p]$, but there exists no tuple t_n in the instance I_n of schema R_n such that $t_n[Y] = t_1[X]$ and $t_n[Y_p] = t_{p_\psi}[Y_p]$. If we choose the instantiation $\rho_1(X_f) = t_1[X_f]$ at step (a), and choose the instantiations $\rho_j[V_f]$ at step (c) based on the instance D, then for each combination of the list P of pointers, there exists no tuple t_n in the instance I_n of schema R_n such that $t_n[Y] = t_1[X]$ and $t_n[Y_p] = t_{p_\psi}[Y_p]$. Thus the algorithm must stop and return 'yes'.

Conversely, assume that the algorithm returns 'yes'. We construct a nonempty database instance D of $\mathcal{R}$ such that $D \models \Sigma$ but $D \not\models \psi$. Let D be the union of all database instances D_P at step (c), where $D_P \models \Sigma_{P,\psi}$. Then as observed earlier, $D \models \Sigma$ and $D \not\models \psi$.

(2) We next show that the problem is PSPACE-hard by reduction from the Q3SAT problem, which is PSPACE-complete (cf. [Pap94]).

An instance of Q3SAT is a first-order logic sentence $\theta = \forall x_1 \exists x_2 \forall x_3 \cdots Q_m x_m \phi$, where Q_m is $\forall$ if m is odd and it is $\exists$ if m is even; $\phi = C_1 \wedge \cdots \wedge C_n$ is an instance of the 3SAT problem in which all the variables are $x_1, \ldots, x_m$, and for each $j \in [1, n]$, the clause C_j is $y_{j_1} \vee y_{j_2} \vee y_{j_3}$ such that for $i \in [1, 3]$, y_{j_i} is either $x_{p_{ji}}$ or $\overline{x_{p_{ji}}}$ for $p_{ji} \in [1, m]$. Here we use $x_{p_{ji}}$ to denote the occurrence of a variable in the literal l_i of clause C_j. The Q3SAT problem is to decide whether θ is satisfiable.

Given an instance θ of Q3SAT, we construct an instance of the implication problem for acyclic CINDs, which consists of a database schema $\mathcal{R}$ with finite-domain attributes,

a set Σ of *acyclic* CINDs defined on $\mathcal{R}$ and another CIND ψ on $\mathcal{R}$. We show that $\Sigma \not\models \psi$ if and only if θ is satisfiable. This suffices. For it holds, then by Immerman–Szelepcsényi theorem [Imm88, Sze87], it is also PSPACE-hard to decide whether $\Sigma \models \psi$.

(a) The database schema $\mathcal{R}$ consists of $m+2$ relation schemas $R_0(A_1,\ldots,A_m)$, …, $R_m(A_1,\ldots,A_m)$, and $S(B)$. All the attributes in $\mathcal{R}$ have a finite domain $\{0,1\}$. Intuitively, each R_i is to encode a quantifier in θ, which is either $\forall$ or $\exists$, for each $i \in [1,m]$. In an instance I_m of schema R_m, each tuple $t[A_1,\ldots,A_m]$ is to carry a truth assignment of the variables $\{x_1,\ldots,x_m\}$ in θ. In addition, we shall use an instance of S to indicate whether θ is satisfied.

(b) The set Σ consists of the following *acyclic* CINDs.

- For each *odd* number $1 \le i \le m$, we define two CINDs $\psi_{i,0}$ and $\psi_{i,1}$ from R_{i-1} to R_i:

  $$\psi_{i,0} = (R_{i-1}[A_1,\ldots,A_{i-1};\mathsf{nil}] \subseteq R_i[A_1,\ldots,A_{i-1};A_i],(\mathsf{nil} \parallel 0)),$$
  $$\psi_{i,1} = (R_{i-1}[A_1,\ldots,A_{i-1};\mathsf{nil}] \subseteq R_i[A_1,\ldots,A_{i-1};A_i],(\mathsf{nil} \parallel 1)).$$

  When $i=1$, $R_{i-1}[A_1,\ldots,A_{i-1}]$ is $R_0[\mathsf{nil}]$. These CINDs assert that for each tuple t in an R_{i-1} relation ($i \in \{1,3,\ldots,m-1\}$), there exist two tuples t_0 and t_1 in the R_i relation such that $t_0[A_1,\ldots,A_{i-1}] = t_1[A_1,\ldots,A_{i-1}] = t[A_1,\ldots,A_{i-1}]$, while $t_0[A_i] = 0$ and $t_1[A_i] = 1$. Intuitively, we encode a universal quantifier $\forall$ by using these CINDs.

- For each *even* number $1 < i \le m$, we define a CIND ψ_i from R_{i-1} to R_i:

  $$\psi_i = (R_{i-1}[A_1,\ldots,A_{i-1};\mathsf{nil}] \subseteq R_i[A_1,\ldots,A_{i-1};\mathsf{nil}],(\mathsf{nil} \parallel \mathsf{nil})).$$

  This CIND ensures that for each tuple t in an R_{i-1} relation ($i \in \{2,4,\ldots,m-2\}$), there exists a tuples t' in the R_i relation such that $t'[A_1,\ldots,A_{i-1}] = t[A_1,\ldots,A_{i-1}]$ while $t'[A_i]$ is either 0 or 1. Intuitively, we encode an existential quantifier $\exists$ by using such CINDs.

- For each clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ in the 3SAT instance ϕ, we define CIND $\psi_{S,j}$ from R_m to S:

  $$\psi_{S,j} = (R_m[\mathsf{nil};A_{p_{j1}},A_{p_{j2}},A_{p_{j3}}] \subseteq S[\mathsf{nil};B],t_{p_{\psi_{m,j}}}),$$

  where $t_{p_{\psi_{m,j}}}[B] = 0$, and for each $i \in [1,3]$, $t_{p_{\psi_{S,j}}}[A_{p_{ji}}] = \xi_j(x_{p_{ji}})$. Here ξ_j is the unique truth assignment of the 3SAT instance ϕ that makes clause C_j false, and $\xi_j(x_{p_{ji}})$ is the truth value of variable $x_{p_{ji}}$ by treating true as 1 and false as 0.

  Intuitively, these CINDs assure that for each tuple t in an R_m relation, $t(A_1,\ldots,A_m)$ denotes a truth assignment ξ for the 3SAT instance ϕ, such that

(a) The CINDs from R_0 to R_1:

$\psi_{1,0} = (R_0[\mathsf{nil};\mathsf{nil}] \subseteq R_1[\mathsf{nill};A_1], (\mathsf{nil} \parallel 0))$,

$\psi_{1,1} = (R_0[\mathsf{nil};\mathsf{nil}] \subseteq R_1[\mathsf{nill};A_1], (\mathsf{nil} \parallel 1))$.

(b) The CIND from R_1 to R_2:

$\psi_2 = (R_1[A_1;\mathsf{nil}] \subseteq R_2[A_1;\mathsf{nil}], (\mathsf{nil} \parallel \mathsf{nil}))$.

(c) The CINDs from R_2 to R_3:

$\psi_{3,0} = (R_2[A_1,A_2;\mathsf{nil}] \subseteq R_3[A_1,A_2;A_3], (\mathsf{nil} \parallel 0))$,

$\psi_{3,1} = (R_2[A_1,A_2;\mathsf{nil}] \subseteq R_3[A_1,A_2;A_3], (\mathsf{nil} \parallel 1))$.

(d) The CIND from R_3 to R_4:

$\psi_4 = (R_3[A_1,A_2,A_3;\mathsf{nil}] \subseteq R_4[A_1,A_2,A_3;\mathsf{nil}], (\mathsf{nil} \parallel \mathsf{nil}))$.

(e) The CINDs from R_4 to S:

$\psi_{S,1} = (R_m[\mathsf{nil};A_1,A_2,A_3] \subseteq S[\mathsf{nil};B], (0,0,0 \parallel 0))$,

$\psi_{S,2} = (R_m[\mathsf{nil};A_2,A_3,A_4] \subseteq S[\mathsf{nil};B], (0,1,0 \parallel 0))$.

Figure 2.7: A (partial) example reduction for the proof of Theorem 2.4.3

for each $i \in [1,m]$, $\xi(x_i) = \mathsf{true}$ if $t[A_i] = 1$, and $\xi(x_i) = \mathsf{false}$ if $t[A_i] = 0$. If the truth assignment ξ makes ϕ false, then there exists a tuple t' in relation S such that $t'[B] = 0$.

The set Σ has no more than $2m+n$ of CINDs in total. Note that Σ is acyclic.

For example, consider the following instance of the Q3SAT problem: $\theta = \forall x_1 \exists x_2 \forall x_3 \exists x_4 C_1 \wedge C_2$, where $C_1 = x_1 \vee x_2 \vee x_3$, and $C_2 = x_2 \vee \bar{x_3} \vee x_4$. Then the set Σ for θ consists of 7 CINDs, as shown in Fig. 2.7.

(c) The CIND $\psi = (R_0[\mathsf{nil};\mathsf{nil}] \subseteq S[\mathsf{nil};B], (\mathsf{nil} \parallel 0))$.

The CIND ψ ensures that if the R_0 relation is not *empty*, then there exists a tuple t in relation S such that $t[B] = 0$.

We next show that the Q3SAT instance θ is satisfiable if and only if $\Sigma \not\models \psi$. First, assume that θ is satisfiable. We define an instance D of $\mathcal{R}$ as follows. For $i \in [1,m]$, the instance I_i of R_i in D consists of all truth assignments for $x_1,\ldots,x_m$ that satisfy the 3SAT instance ϕ. The instance I_S of S in D consists of a single tuple s with $s[B] = 1$. One can readily verify that $D \models \Sigma$ but $D \not\models \psi$. Hence $\Sigma \not\models \psi$.

Conversely, assume that $\Sigma \not\models \psi$. Then there exists an instance D of $\mathcal{R}$ such that $D \models \Sigma$ but $D \not\models \psi$. By $D \not\models \psi$, the instance of I_0 of schema R_0 in D is nonempty, and hence so is the instance I_i of R_i in D for all $i \in [1,m]$, by $D \models \Sigma$. Observe that the

instance I_m of R_m encodes truth assignments for $x_1, \ldots, x_m$. By $D \models \Sigma$, we know that I_{m-1} includes all the truth assignments required by the quantifiers in θ. By $D \not\models \psi$ again, the instance I_S of S in D does not have any tuple s with $s[B] = 0$. Hence by the definition of the CINDs $\psi_{m,j}$, none of those truth assignments in I_m violates the 3SAT instance ϕ. Therefore, θ is satisfiable. □

2.5 Unary Conditional Inclusion Dependencies

Another well-studied fragment of INDs is the class of unary inclusion dependencies, defined as follows [AHV95, CKV90].

A *unary* IND (UIND) is an IND of the form $R_a[A] \subseteq R_b[B]$, where $A \in \mathsf{attr}(R_a)$ and $B \in \mathsf{attr}(R_b)$. That is, a UIND is an IND in which exactly one attribute appears on each side.

In this section we define and investigate unary CINDs.

Unary CINDs. A *unary* CIND (UCIND) is a CIND of the form $(R_a[A; X_p] \subseteq R_b[B; Y_p], t_p)$, where A and B are attributes in R_a and R_b, respectively.

That is, UCINDs are an extension of UINDs by incorporating patterns of data values. Observe that patterns X_p and Y_p in a UCIND may have more than one attribute.

It is common to find UCINDs in practice.

Example 2.5.1: Consider a database consisting of three relations: $\mathsf{student}(\mathsf{SSN}, \mathsf{name}, \mathsf{dept})$, $\mathsf{course}(\mathsf{cno}, \mathsf{title}, \mathsf{dept})$, and $\mathsf{enroll}(\mathsf{SSN}, \mathsf{cno}, \mathsf{grade})$. The student relation collects all the student records in a university, and course consists of all the courses offered by the university. In contrast, the enroll relation aims to maintain a complete record of the CS courses registered by students in the CS department.

One would naturally want the following CINDs:

$(\mathsf{student}[\mathsf{SSN}; \mathsf{dept}] \subseteq \mathsf{enroll}[\mathsf{SSN}; \mathsf{nil}], (\mathsf{student}[\mathsf{dept}] = \text{`CS'} \parallel \mathsf{nil}),$
$(\mathsf{course}[\mathsf{cno}; \mathsf{dept}] \subseteq \mathsf{enroll}[\mathsf{cno}; \mathsf{nil}], (\mathsf{course}[\mathsf{dept}] = \text{`CS'} \parallel \mathsf{nil}),$
$(\mathsf{enroll}[\mathsf{SSN}; \mathsf{nil}] \subseteq \mathsf{student}[\mathsf{SSN}; \mathsf{nil}], \emptyset)$, and
$(\mathsf{enroll}[\mathsf{cno}; \mathsf{nil}] \subseteq \mathsf{course}[\mathsf{cno}; \mathsf{nil}], \emptyset)$.

All these CINDs are UCINDs. □

We next investigate the static analysis of UCINDs. By Theorem 2.3.1 we do not have to worry about the satisfiability problem for UCINDs. Hence below we focus on the implication problem for UCINDs, *i.e.,* the problem for determining, given a set Σ of

UCINDs and another UCIND φ, whether $\Sigma \models \varphi$. We study the problem in the absence of finite-domain attributes and in the general setting.

In the absence of finite-domain attributes. It has been shown that the implication problem for UINDs is in polynomial time (PTIME) in this setting [CKV90]. We next show that the implication analysis of UCINDs can also be conducted efficiently when finite-domain attributes are not present.

Theorem 2.5.1 *The implication problem for* UCINDs *is in polynomial time in the absence of finite-domain attributes.*

Proof: It suffices to give a PTIME algorithm for checking whether $\Sigma \models \psi$ or not. Similar to the *upper bound* proof of Theorem 2.3.10, the algorithm converts the problem to the graph reachability problem, *i.e.,* checking whether there exists a path from a node to another in a graph. Recall that the graph reachability problem is solvable in quadratic time [Pap94].

The proof consists of three parts. We first present the algorithm. We then show that the algorithm is correct. Finally, we show that the algorithm is in PTIME. Consider a set $\Sigma \cup \{\psi\}$ of UCINDs over a database schema $\mathcal{R} = (R_1, \ldots, R_n)$, where the UCIND $\psi = (R_a[A; X_p] \subseteq R_b[B; Y_p], t_{p_\psi})$.

(1) The algorithm simulates the chase procedure given in the proof of Theorem 2.3.2, fine-tuned to leverage unary CINDs.

(a) The algorithm first builds a directed graph $G(V, E)$, based on which it then checks whether $\Sigma \models \psi$.

A node in the graph G is in the form of $(R_i[C; U_p], t[U_p])$ $(1 \leq i \leq n)$ such that (a) R_i is a schema in $\mathcal{R}$, (b) C is a single attribute in $\mathsf{attr}(R_i)$, (c) U_p is a list of attributes in $\mathsf{attr}(R_i)$, and (d) $t[U_p]$ is a partial tuple of R_i defined on U_p only.

The set V of nodes in G includes the following: (a) a single node $\mathsf{u_a} = (R_a[A; X_p], t_{p_\psi}[X_p])$, which corresponds to the LHS of the UCIND ψ; and (b) for each UCIND $\psi' = (R_i[C; U_p] \subseteq R_j[F; V_p], t_p)$ in Σ, a node $\mathsf{u} = (R_j[F; V_p], t_p[V_p])$, which denotes the RHS of the UCIND ψ'.

The set E of edges contains a directed edge $(\mathsf{u}_1, \mathsf{u}_2)$ for all nodes $\mathsf{u}_1 = (R_i[C; U_p], t_i[U_p])$ and $\mathsf{u}_2 = (R_j[F; V_p], t_j[V_p])$ in V if there is a UCIND $(R_i[C; U'_p] \subseteq R_j[F; V'_p], t_p)$ in Σ such that $U'_p \subseteq Z_p$, $V_p \subseteq V'_p$, $t_p[U'_p] = t_i[U'_p]$, and $t_p[V_p] = t_j[V_p]$.

(b) The algorithm then checks whether $\Sigma \models \psi$, based on G.

Let S_b be the set of nodes that are of the form $\mathsf{v} = (R_b[B; U_p], t_b[U_p])$ in G such that $Y_p \subseteq Z_p$ and $t_b[Y_p] = t_{p_\psi}[Y_p]$. Recall that Y_p and $t_{p_\psi}[Y_p]$ are from the UCIND ψ.

The algorithm checks whether there exists a node u in the node set S_b such that there is a path from the node u_a to u in the graph G (recall that u_a denotes the LHS of ψ). If there exists such u, it returns 'yes', and it returns 'no' otherwise.

(2) We now verify the correctness of the algorithm, *i.e.,* the algorithm returns 'yes' iff $\Sigma \models \psi$.

First assume that the algorithm returns 'yes'. Then there must exist a path from the node u_a to a node $\mathsf{u}_\mathsf{b} = (R_b[B; U_p], t_b[U_p])$ in the graph G, where $Y_p \subseteq Z_p$ and $t_b[Y_p] = t_{p_\psi}[Y_p]$. Along the same lines as the proof of Theorem 2.3.2, one can construct a proof to show that $\Sigma \vdash_{\mathcal{I}(1-6)} \psi$ based on CIND1–CIND6 in the inference system $\mathcal{I}$ (see Section. 2.3.2.1). By Theorem 2.3.2, CIND1–CIND6 are sound and complete for the implication analysis of CINDs in the absence of finite-domain attributes. Hence $\Sigma \models \psi$.

Conversely, assume that the algorithm returns 'no'. We show that $\Sigma \not\models \psi$ by constructing a database instance D of $\mathcal{R}$ such that $D \models \Sigma$, but $D \not\models \psi$. The instance D is constructed step by step as follows.

(a) Initialize $D = \{I_1, \ldots, I_n\}$ such that $I_1 = \ldots = I_n = \emptyset$.

(b) Create a tuple t_a such that $t_a[A] = v$, $t_a[X_p] = t_{p_\psi}[X_p]$, and $t_a[A'] = v_0$ for all the other attributes A' in $\mathtt{attr}(R_a)$, and let $I_a = I_a \cup \{t_a\}$. Here v and v_0 are two distinct variables.

(c) For each node $\mathsf{u} = (R_i[C; U_p], t_i[U_p])$ in the graph G such that there exists a path from nodes u_a to u, construct a tuple t such that $t[C] = v$, $t[U_p] = t_i[U_p]$, and $t[C'] = v_0$ for all the other attributes C' in $\mathtt{attr}(R_i)$, and let $I_i = I_i \cup \{t\}$.

(d) Extend the instance D by using the chase procedure given in the proof of Theorem 2.3.2 until D reaches a fixpoint.

Then as argued in the proof of Theorem 2.3.2 about the chase procedure, one can verify that $D \models \Sigma$, but $D \not\models \psi$. Therefore, $\Sigma \not\models \psi$.

(3) We next show that the algorithm is in polynomial time.

It is obvious that the graph G can be built in polynomial time. Observe that the size of the set S_b is bounded by the number nodes of the graph G, of which the size is bounded by a polynomial in the size of $\Sigma \cup \{\psi\}$. Based on these, it is easy to verify that the algorithm is indeed in PTIME.

Putting these together, we conclude that in the absence of finite-domain attributes, the implication problem for UCINDs is solvable in PTIME. □

In the general setting. For UINDs, the presence of finite-domain attributes does not complicate the implication analysis. Indeed, a close examination of the PTIME algorithm of [CKV90] for checking UIND implication reveals that it still works in the general setting. Hence we have the following.

Corollary 2.5.2 *The implication problem for* UINDs *remains in polynomial time in the general setting.*

This is, however, no longer the case for UCINDs.

Theorem 2.5.3 *The implication problem for* UCINDs *is coNP-complete in the general setting.*

Proof: (1) We first show that the problem is in coNP. Consider a set $\Sigma \cup \{\psi\}$ of UCINDs defined on a database schema $\mathcal{R} = (R_1, \ldots, R_n)$. To show that the problem is in coNP, it suffices to give NP algorithms for checking whether $\Sigma \not\models \psi$.

We first show that UCINDs can be transformed into two normal forms. We then present two NP algorithms based on the form of ψ, and show that the algorithms are correct.

(a) We first show that UCINDs can be expressed in certain "normal" forms. Consider a UCIND $\varphi = (R_i[C;U_p] \subseteq R_j[F;V_p],\ t_{p_\varphi})$ such that the attribute C has a finite domain $\mathsf{dom}(C) = \{c_1, \ldots, c_k\}$. Let $\Sigma_\varphi = \{\varphi_1, \ldots, \varphi_k\}$, where for each $l \in [1,k]$, $\varphi_l = (R_i[\mathsf{nil};C,U_p] \subseteq R_j[\mathsf{nil};F,V_p], t_{p_{\varphi_l}})$ such that $t_{p_{\varphi_l}}[U_p \parallel V_p] = t_{p_\varphi}[U_p \parallel V_p]$ and $t_{p_{\varphi_l}}[C] = t_{p_{\varphi_l}}[F] = `c_l'$. It is easy to verify that $\Sigma_\varphi \equiv \{\varphi\}$, by CIND4 and CIND8 in the inference system $\mathcal{I}$ for CINDs (see Section. 2.3.2.1).

As a result, given a set Σ of UCINDs, we can derive an equivalent set Σ' of CINDs by using the transformations above. Note that the number of CINDs in Σ' is bounded by a *polynomial* of the size of $\mathcal{R}$ and the number of UCINDs in Σ. Hence we can assume *w.l.o.g.* that all the UCINDs in $\Sigma \cup \{\psi\}$ are of one of the following forms:

- $(R_i[\mathsf{nil};U_p] \subseteq R_j[\mathsf{nil};V_p], t_p)$; and
- $(R_i[C;U_p] \subseteq R_j[F;V_p], t_p)$, where both attributes C and F have an infinite domain.

Given a set $\Sigma \cup \{\psi\}$ of CINDs in these two forms, we develop two NP algorithms for checking whether $\Sigma \not\models \psi$, depending on the form of the CIND ψ. We use Σ_1 and Σ_2 to denote those CINDs in Σ in the first form and those in the second one, respectively, where $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$.

(b) We now provide an NP algorithm for the first case where the CIND ψ is $(R_a[\mathsf{nil};X_p] \subseteq R_b[\mathsf{nil};Y_p], t_{p_\psi})$.

To treat Σ_1 and Σ_2 uniformly, we further transform the CINDs of Σ_2 into CINDs in the first form. More specifically, for each CIND $\varphi = (R_i[C;U_p] \subseteq R_j[F;V_p], t_{p_\varphi})$ in Σ_2, we define a set Σ_φ of CINDs in the first form as follows.

- Let adom = $\{a_1, \ldots, a_h\}$ be the set of constants appearing in either $\Sigma \cup \{\psi\}$ or in the finite domains of $\mathcal{R}$.
- Define Σ_φ to be the set $\{\varphi_0, \varphi_1, \ldots, \varphi_h\}$, where $\varphi_0 = (R_i[\mathsf{nil}; U_p] \subseteq R_j[\mathsf{nil}; V_p], t_{p_\varphi})$, and for each $l \in [1, h]$, $\varphi_l = (R_i[\mathsf{nil}; C, U_p] \subseteq R_j[\mathsf{nil}; F, V_p], t_{p_{\varphi_l}})$ such that $t_{p_{\varphi_l}}[U_p \parallel V_p] = t_{p_\varphi}[U_p \parallel V_p]$ and $t_{p_{\varphi_l}}[C] = t_{p_{\varphi_l}}[F] = `a_l'$.
  Here CIND φ_0 is derived from φ by the rule CIND2, and the other CINDs in Σ_φ are derived from φ by CIND4 in the inference system $\mathcal{I}$ for CINDs.

Observe that the number of CINDs in Σ_φ is bounded by a polynomial of the size of $\mathcal{R}$, Σ, and ψ.

Let Σ_2' be the union of Σ_φ's when φ ranges over all CINDs in Σ_2. We show that it suffices to consider CINDs in $\Sigma_1 \cup \Sigma_2'$. Indeed, a close examination of the chase procedure given in the proof of Theorem 2.3.4 tells us that the chase process for $\Sigma_1 \cup \Sigma_2 \cup \{\psi\}$ is equivalent to the one for $\Sigma_1 \cup \Sigma_2' \cup \{\psi\}$ since whenever a CIND in $\Sigma_1 \cup \Sigma_2$ is applied, we can use one in $\Sigma_1 \cup \Sigma_2'$ instead to reach the same result, and vice versa. Recall that the chase procedure is used to decide whether $\Sigma \models \psi$ for CINDs in the general setting.

We now give the details of the NP algorithm, which is a non-deterministic extension of the PTIME algorithm given in the proof of Theorem 2.5.1, to handle finite-domain attributes. More specifically, given the set $\Sigma_1 \cup \Sigma_2' \cup \{\psi\}$ of CINDs, it extends the PTIME algorithm as follows.

- It adds an extra step after generating the node set V but before generating the edge set E. For each node $\mathsf{u} = (R_i[\mathsf{nill}; U_p], t[U_p])$ in V, it *guesses* an instantiation ρ_u for the list U_f of all finite-domain attributes in $\mathsf{attr}(R_i) \setminus (U_p)$ such that for each attribute $C' \in U_f$, $\rho_u(C')$ is a data value drawn from the finite $\mathsf{dom}(C')$. Note that the format of nodes is a little different from those used in the PTIME algorithm. However, this has no impact on the algorithm itself.
- The NP algorithm returns an answer *opposite* to that of the PTIME algorithm.
  That is, if there exists *no* node $\mathsf{u_b}$ in the set S_b such that there exists a path from the node $\mathsf{u_a}$ to $\mathsf{u_b}$ in the graph G, the algorithm returns 'yes', and it returns 'no' otherwise. Recall that $\mathsf{u_a}$ and $\mathsf{u_b}$ denote the LHS and the RHS of ψ, respectively. This is because the NP algorithm checks whether $\Sigma \not\models \psi$, while the PTIME algorithm checks whether $\Sigma \models \psi$.

It is easy to see that this is an NP algorithm.

We next show that algorithm returns 'yes' if and only if $\Sigma \not\models \psi$. Assume first that the NP algorithm returns 'yes'. Along the same lines as the argument for the 'no' answer of

the PTIME algorithm given in the proof of Theorem 2.5.1, we can construct a nonempty instance D of $\mathcal{R}$ such that $D \models \Sigma$ but $D \not\models \psi$, *i.e.,* $\Sigma \not\models \psi$. Here when populating D, it suffices to randomly *guess* an instantiation for the finite-domain attributes of the new tuple to be inserted into D (the one shown in the NP algorithm will do).

Conversely, assume that $\Sigma \not\models \psi$. We show that there exists a set of instantiations for the node set V such that the algorithm returns 'yes'. Since $\Sigma \not\models \psi$, there exists a database instance $D = (I_1, \ldots, I_n)$ such that $D \models \Sigma$ and $D \not\models \psi$. That is, there exists a tuple $t_a \in I_a$ such that $t_a[X_p] = t_{p_\psi}[X_p]$, but there exists no tuple $t_b \in I_b$ such that $t_b[Y_p] = t_{p_\psi}[Y_p]$. The instantiations are defined as follows. For the node $\mathsf{u_a} = (R_a[\mathsf{nill}; X_p], t_{p_\psi}[X_p])$ in V, define an instantiation ρ_{u_a} such that $\rho_{u_a}[X_f] = t_a[X_f]$ for the list X_f of all finite-domain attributes in $\mathtt{attr}(R_a) \setminus X_p$. For all the other nodes $\mathsf{u} = (R_i[\mathsf{nil}; U_p], t[U_p])$ in V, if there exists a tuple $t_u \in I_i$ such that $t_u[U_p] = t[U_p]$, define an instantiation ρ_u such that $\rho_u[U_f] = t_u[U_f]$ for the list U_f of all finite-domain attributes in $\mathtt{attr}(R_u) \setminus U_p$. Otherwise, let $\rho_u[U_f]$ be defined in terms of arbitrary values in the domains. The algorithm must return 'yes' for this specific graph G since there exist no tuples $t_b \in I_b$ such that $t_b[Y_p] = t_{p_\psi}[Y_p]$.

(c) We next present an NP algorithm for the second case where the CIND ψ is of the form $(R_a[A; X_p] \subseteq R_b[B; Y_p], t_{p_\psi})$.

This case is simpler. Indeed, the chase procedure given in the proof of Theorem 2.3.4 tells us that those CINDs in Σ_1 can be simply left out. In this case, the NP algorithm is the same as the one for the first case, except that only those CINDs in Σ_2 are considered when generating the graph G. Here the nodes have the same format as those used in the PTIME algorithm given in the proof of Theorem 2.5.1.

An argument similar to the one for the first case can verify that this NP algorithm returns 'yes' if and only if $\Sigma \not\models \psi$.

(2) We next show that the problem is coNP-hard by reduction from the 3SAT problem to the complement of the problem (*i.e.,* to decide whether $\Sigma \not\models \psi$). It is known that 3SAT is NP-complete (cf. [GJ79]).

Consider an instance $\phi = C_1 \wedge \cdots \wedge C_n$ of 3SAT, where $x_1, \ldots, x_m$ are all the variables in ϕ, and for each $j \in [1, n]$, C_j is of the form $y_{j_1} \vee y_{j_2} \vee y_{j_3}$ such that for $i \in [1, 3]$, y_{j_i} is either $x_{p_{ji}}$ or $\overline{x_{p_{ji}}}$ for $p_{ji} \in [1, m]$. Here $x_{p_{ji}}$ denotes the occurrence of a variable in the literal l_i of clause C_j.

Given an instance ϕ of 3SAT, we construct an instance of the implication problem for unary CINDs, which consists of a database schema $\mathcal{R}$ with finite-domain attributes,

(a) The database schema $\mathcal{R} = (R(B, A_1, \ldots, A_5), S(B, C))$.

(b) The set of UCINDs $\Sigma = \{\varphi_1, \varphi_2, \varphi_3\}$, where
$\varphi_1 = (R[B; A_1, A_2, A_3] \subseteq S[B; C], (0, 0, 0 \parallel 0))$,
$\varphi_2 = (R[B; A_2, A_3, A_4] \subseteq S[B; C], (0, 1, 0 \parallel 0))$, and
$\varphi_3 = (R[B; A_3, A_4, A_5] \subseteq S[B; C], (0, 1, 1 \parallel 0))$.

(c) The UCIND $\psi = (R[B; \mathsf{nill}] \subseteq S[B; C], (\mathsf{nill} \parallel 0))$.

Figure 2.8: An example reduction for the proof of Theorem 2.5.3

and a set $\Sigma \cup \{\psi\}$ of UCINDs defined on $\mathcal{R}$. We show that $\Sigma \not\models \psi$ if and only if ϕ is satisfiable.

(a) The database schema $\mathcal{R}$ consists of two relation schemas $R(B, A_1, \ldots, A_m)$ and $S(B, C)$, in which all attributes have a finite domain $\{0, 1\}$.

Intuitively, a tuple $t(A_1, \ldots, A_m)$ in an instance I_R of schema R denotes a truth assignment ξ_t of the 3SAT instance ϕ, such that for each $i \in [1, m]$ $\xi_t(x_i) = \mathsf{true}$ if $t[A_i] = 1$, and $\xi_t(x_i) = \mathsf{false}$ if $t[A_i] = 0$. As will be seen shortly, an instance I_S of schema S is used to indicate whether ϕ is satisfiable.

(b) The set Σ consists of n UCINDs given as follows. For each $j \in [1, n]$, let ξ_j be the *unique* truth assignment that makes the clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ false. Then we define a UCIND $\varphi_j = (R[B; A_{p_{j1}}, A_{p_{j2}}, A_{p_{j3}}] \subseteq S[B; C], t_{p_{\varphi_j}})$, where $t_{p_{\varphi_j}}[C] = 0$, and for each $i \in [1, 3]$, $t_{p_{\varphi_j}}[A_{p_{ji}}] = 1$ if $\xi_j(x_{p_{ji}}) = \mathsf{true}$, and $t_{p_{\varphi_j}}[A_{p_{ji}}] = 0$ otherwise.

These UCINDs assert that for a tuple t in an R relation, if its carries a truth assignment ξ_t that makes ϕ false, then there must exist a tuple t' in the S relation such that $t'[B] = t[B]$ and $t'[C] = 0$. Observe that if there exists a database instance $D = (I_R, I_S)$ such that I_R is nonempty, I_S is empty, and $D \models \Sigma$, then the 3SAT instance ϕ must be satisfiable.

(c) The UCIND $\psi = (R[B; \mathsf{nil}] \subseteq S[B; C], (\mathsf{nil} \parallel 0))$. It enforces that for each tuple t in relation I_R, there exists a tuple t' in relation I_S such that $t'[B] = t[B]$ and $t'[C] = 0$.

As an example, consider an instance $\phi = C_1 \wedge C_2 \wedge C_3$ of the 3SAT problem, where $C_1 = x_1 \vee x_2 \vee x_3$, $C_2 = x_2 \vee \bar{x_3} \vee x_4$ and $C_3 = x_3 \vee \bar{x_4} \vee \bar{x_5}$. The reduction for ϕ is shown in Fig. 2.8.

The reduction is obviously in polynomial time.

We next show that $\Sigma \not\models \psi$ if and only if the 3SAT instance ϕ is satisfiable. We first assume that ϕ is satisfiable, and show that $\Sigma \not\models \psi$. It suffices to construct a database D such that $D \models \Sigma$ but $D \not\models \psi$. Since ϕ is satisfiable, there exists a truth assignment ξ that

satisfies ϕ. Based on ξ, we define a tuple t on R such that (a) for each $i \in [1,m]$, $t[A_i] = 1$ if $\xi_t(x_i) = \mathsf{true}$, and $t[A_i] = 0$ if $\xi_t(x_i) = \mathsf{false}$, and (b) $t[B] = 1$. Let the instance $D = (I_R, I_S)$ such that $I_R = \{t\}$ and $I_S = \emptyset$. Then $D \models \Sigma$ but $D \not\models \psi$.

Conversely, we assume that $\Sigma \not\models \psi$, and show that ϕ is satisfiable. It suffices to find a truth assignment ξ that satisfies ϕ. Since $\Sigma \not\models \psi$, there exists a database instance $D = (I_R, I_S)$ such that $D \models \Sigma$ but $D \not\models \psi$. By $D \not\models \psi$, there is a tuple t in I_R but there exists no tuple t' in I_S with $t'[B] = t[B]$ and $t'[C] = 0$. Define a truth assignment ξ for ϕ such that for each $i \in [1,m]$, $\xi(x_i) = \mathsf{true}$ if $t[A_i] = 1$, and $\xi(x_i) = \mathsf{false}$ if $t[A_i] = 0$. Then ξ satisfies ϕ since otherwise, there must exist a tuple t' in I_S with $t'[B] = t[B]$ and $t'[C] = 0$ by the definition of those UCINDs in Σ. Hence ϕ is satisfiable. □

Acyclic UCINDs. One might be tempted to think that it would simplify the implication analysis if we further restrict UCINDs to be acyclic. Unfortunately, this is not the case. Indeed, in the lower bound proof for Theorem 2.5.3, the UCINDs used are *acyclic*. From this it follows that the implication problem for acyclic UCINDs remains to be coNP-complete.

Corollary 2.5.4 *The implication problem for acyclic* UCIND*s is coNP-complete in the general setting.*

2.6 Related Work

Data dependencies have been studied for relational databases since the introduction of FDs by Codd [Cod72] in 1972 (see, *e.g.,* [AHV95, FV84] for details). Recently, data dependencies have generated renewed interests for improving data quality ([ABC03b, Ber06, Cho07, Fan08]) and for schema mapping ([BEFF06, HHH+05, Kol05b]).

The theory of INDs was established in [CFP84], which developed a sound and complete inference system and the PSPACE-completeness for the implication analysis of INDs. Acyclic INDs were introduced in [Sci86], and their implication problem was shown to be NP-complete in [CK84]. Unary INDs were studied in [CKV90], which provided a sound and complete inference system for UINDs and FDs, and proved the PTIME bound of the implication problem for UINDs and FDs put together (see [AHV95] for a survey on INDs, AINDs and UINDs). While not explicitly stated, the proofs of these results indicate that the implication analysis was conducted in the absence of finite-domain attributes. In this work we verify that the complexity bounds for INDs, AINDs and UINDs remain intact in the presence of finite-domain attributes.

CINDs, ACINDs and UCINDs extend INDs, AINDs and UINDs, respectively, by incorporating patterns of data values. For the implication problem in the absence of finite-domain attributes, the lower bounds for CINDs, AINDs and UINDs are inherited from their traditional counterparts, but *not* the upper bounds. When finite-domain attributes may be present, however, none of the results of [BV84, CFP84, CK84, CKV90, Sci86] holds on CINDs. Indeed, the implication problems for CINDs, ACINDs and UCINDs in the general setting have a higher complexity bound than their traditional counterparts.

INDs are a special case of TGDs, which can be expressed as first-order logic sentences of the form:

$$\forall x_1 \ldots \forall x_n[\varphi(x_1, \ldots, x_n) \rightarrow \exists z_1 \ldots \exists z_k \phi(y_1, \ldots, y_m)],$$

where (a) $\{z_1, \ldots, z_k\} = \{y_1, \ldots, y_m\} \setminus \{x_1, \ldots, x_n\}$, (b) φ and ϕ are conjunctions of relation atoms of the form $R(w_1, \ldots, w_l)$ in which $w_1, \ldots, w_l$ are variables (see *e.g.,* [AHV95] for details). In contrast to CINDs, TGDs do not allow constants, and their the implication problem is *undecidable* [BV84].

There have been extensions of TGDs [MS96] developed for constraint databases, notably constrained tuple-generating dependencies (CTGDs) of the form:

$$\forall \bar{x}(R_1(\bar{x}) \wedge \ldots R_k(\bar{x}) \wedge \xi \rightarrow \exists \bar{y}(R'_1(\bar{x}, \bar{y}) \wedge \ldots \wedge R'_s(\bar{x}, \bar{y}) \wedge \xi'(\bar{x}, \bar{y})),$$

where R_i, R'_j are relation atoms, and ξ, ξ' are arbitrary constraints. While CTGDs support constants and can express CINDs, the increased expressive power comes at a price for static analysis. Indeed, the satisfiability and implication problems are both undecidable for CTGDs.

Closer to our work is the recent study of CFDs [FGJK08]. CFDs extend FDs with pattern tableaux, along the same lines as CINDs. It was shown in [FGJK08] that the satisfiability and implication problems for CFDs are NP-complete and coNP-complete, respectively, in the general setting, and they are in PTIME in the absence of finite-domain attributes. Extensions of CFDs have been proposed to support disjunction and negation [BFGM08], cardinality constraints and synonym rules [CFM09b], built-in predicates ($\neq, <, \leq, >, \geq$) [CFM09a], and to specify patterns in terms of value ranges [GKK$^+$08]. However, CFDs and their extensions are defined on a single relation and moreover, are universally quantified. They cannot express CINDs, and neither CINDs nor their static analyses were studied in [FGJK08, BFGM08, CFM09b, CFM09a, GKK$^+$08]. In addition, as we have seen earlier, the satisfiability and implication analysis of CINDs are far more intriguing than their CFD counterparts. An

extension of CINDs was recently proposed to support built-in predicates [CFM09a], which was based on the results of this work.

Research on constraint-based data cleaning has mostly focused on two topics, both proposed by [ABC03b]: *repairing* is to find another database that is consistent and minimally differs from the original database (*e.g.,* [BFFR05, CM05, FPL+01]); and *consistent query answering* is to find an answer to a given query in every repair of the original database (*e.g.,* [ABC03b, Wij05]). A variety of constraint formalisms have been used in data cleaning, ranging from standard FDs and INDs [ABC03b, BFFR05, CM05], denial constraints (full dependencies) [LB07], to logic programs (see [Ber06, Cho07, Fan08] for recent surveys). To our knowledge, no prior work has considered CINDs for data cleaning albeit our work [BFM07, CFM09a] remarked earlier. Moreover, previous work on data cleaning did not study inference, satisfiability and implication analysis of constraints, which are the focus of this work.

Constraints used in schema matching in practice are typically standard INDs and keys (see, *e.g.,* [HHH+05]). Contextual schema matching [BEFF06] investigated the applications of contextual foreign keys, a primitive and special case of CINDs, in deriving schema mapping from schema matches. While [BEFF06] partly motivated this work, it neither formalized the notion of CINDs nor considered the static analysis of CINDs. There has also been recent work on data exchange (schema mapping) and data integration based on TGDs (see [APRR09, Kol05b, Len02] for surveys). However, inference systems and static analysis of constraints are not the focus of the work on data exchange and data integration, and none of the results of this work has been established in those lines of research.

The *chase* technique is widely used in implication analysis and query optimization, and has been studied for a variety of dependencies (see, *e.g.,* [AHV95, BV84, CKV90, JK84]). Recently it was extended for query reformulation and schema mapping, and a number of sufficient conditions were identified for its termination (see [DPT06, Kol05b] for recent surveys). This work extends the chase technique to study the implication analysis of CINDs, for which the chase process always terminates.

Chapter 3

Extensions of Conditional Dependencies

In this chapter we introduce extensions of both conditional functional dependencies (CFDs, [FGJK08]) and conditional inclusion dependencies (CINDs, Chapter 2 and [BFM07]) to capture inconsistencies that arise in practice but cannot be detected by CFDs and CINDs.

We first propose an extension of CFDs, referred to as eCFDs, which further specify patterns of semantically related values in terms of disjunction and negation. The increase of expressive power does not incur extra complexity: we show that the satisfiability and implication analysis of eCFDs remain NP-complete and coNP-complete, respectively, *the same as* their CFDs counterparts.

We then propose an extension of CFDs, denoted by CFD^{c}s, to express cardinality constraints, domain-specific conventions, and patterns of semantically related constants in a uniform constraint formalism. And we show that despite the increased expressive power, the satisfiability and implication problems for CFD^{c}s remain NP-complete and coNP-complete, respectively, the same as their counterparts for CFDs.

We finally propose a natural extension of CFDs and CINDs, denoted by CFD^{p}s and CIND^{p}s, respectively, by specifying patterns of data values with $=$, $\neq$, $<$, $\leq$, $>$ and $\geq$ predicates. And we show that despite the increased expressive power, the static analysis of CFD^{p}s and CIND^{p}s retain the same complexity as their CFDs and CINDs counterparts, respectively.

3.1 Introduction

Extensions of functional dependencies (FDs) and inclusion dependencies (INDs), known as *conditional functional dependencies* (CFDs, [FGJK08]) and *conditional inclusion dependencies* (CINDs, Chapter 2 and [BFM07]), respectively, have recently been proposed for improving data quality. These extensions enforce patterns of semantically related data values, and detect errors as violations of the dependencies. Conditional dependencies are able to capture more inconsistencies than FDs and INDs [FGJK08, BFM07], which were currently the basis of many data-cleaning tools [ABC03b, BFFR05, CM05, FPL+01, Wij05].

We first briefly review CFDs. A CFD φ on a relation schema R is a pair $R(X \to Y, t_p)$, where (1) $X \to Y$ is a standard FD, called the FD *embedded in* φ; and (2) t_p is a tuple with attributes in X and Y, referred to as the *pattern tuple* of φ, where for each A in X (or Y), $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed variable '_' that draws values from $\mathsf{dom}(A)$. We separate the X and Y attributes in t_p with '$\|$'.

Example 3.1.1: Some example CFDs taken from [FGJK08] are given below:

φ_1: $R([\mathsf{CC}, \mathsf{zip}] \to [\mathsf{street}],\ (44, \_ \,\|\, \_))$,
φ_2: $R([\mathsf{CC}, \mathsf{AC}] \to [\mathsf{city}],\ (44, \_ \,\|\, \_))$,
φ_4: $R([\mathsf{CC}, \mathsf{AC}] \to [\mathsf{city}],\ (44, 20 \,\|\, \text{LDN}))$,
f_1: $R_1(\mathsf{zip} \to \mathsf{street},\ (\_ \,\|\, \_))$.

The standard FD f_1 on source R_1 is expressed as a CFD. □

The semantics of CFDs is defined in terms of a relation $\asymp$ on constants and '_': $\eta_1 \asymp \eta_2$ if either $\eta_1 = \eta_2$, or one of η_1, η_2 is '_'. The operator $\asymp$ naturally extends to tuples, *e.g.,* (Portland, LDN) $\asymp$ (_, LDN) but (Portland, LDN) $\not\asymp$ (_, NYC). We say that a tuple t_1 *matches* t_2 if $t_1 \asymp t_2$.

An instance D of R *satisfies* $\varphi = R(X \to Y, t_p)$, denoted by $D \models \varphi$, if for *each pair* of tuples t_1, t_2 in D, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$.

Intuitively, φ is a constraint defined on the set $D_\varphi = \{t \mid t \in D, t[X] \asymp t_p[X]\}$ such that for any $t_1, t_2 \in D_\varphi$, if $t_1[X] = t_2[X]$, then (a) $t_1[Y] = t_2[Y]$, and (b) $t_1[Y] \asymp t_p[Y]$. Here (a) enforces the semantics of the embedded FD, and (b) assures the binding between *constants* in $t_p[Y]$ and *constants* in $t_1[Y]$. Note that φ is defined on the subset D_φ of D identified by $t_p[X]$, rather than on the entire D.

We say that an instance D of a relational schema $\mathcal{R}$ satisfies a set Σ of CFDs defined on $\mathcal{R}$, denoted by $D \models \Sigma$, if $D \models \phi$ for each ϕ in Σ.

Both CFDs and CINDs specify constant patterns in terms of equality $(=)$ only. To capture inconsistencies that commonly arise in real-life data, however, one often needs to use more expressive constraints, illustrated by the following examples.

Example 3.1.2: (a) In USA, if a city is Newark, then its associated state must be either Delaware, Ohio, or New Jersey; (b) most cities in the New York state of USA have a *unique* area code, except NYC (New York City) and LI (Long Island); (c) UK and United Kingdom are semantically equivalent, *i.e.,* they refer to the same object; (d) in a school, a student can register for at most six courses each semester; (e) in UK, if one is married, then her/his age is above 15; and (f) in a company, the workers' week salary is in the range of $[1000, 1500]$.

However, neither CFDs nor CINDs can handle these practically needed constraints. To specify (a) and (b), we need disjunctions and negations; to specify (c) and (d), we need cardinality constraints and synonym rules; and to specify (e) and (f), we need other predicates, such as $\neq, <, \leq, >$ and $\geq$. These are beyond the expressive powers of CFDs and CINDs since they only specify constant patterns in terms of equality $(=)$. □

Contributions. (1) The first contribution is three CFD extensions and one CIND extension, and all are well-balanced between expressiveness and complexity.

- The first extension of CFDs, referred to as eCFDs, introduces disjunctions and negations to CFDs.
- The second extension of CFD^{c}s, referred to as CFD^{c}s, is able to express cardinality constraints, synonym rules and patterns of semantically related values of CFDs in a uniform constraint formalism.
- The third extension of CFDs, referred to as CFD^{p}s, extends CFDs by further supporting $\neq, <, \leq, >, \geq$ predicates.
- The extension of CINDs, referred to CIND^{p}s, extends CINDs by further supporting $\neq, <, \leq, >, \geq$ predicates.

Putting these dependencies together, we can deal with all constraints mentioned in Example 3.1.2.

(2) Our second contribution consists of complexity bounds of two central technical problems associated with each class of the proposed data dependencies: the *satisfiability* problem and the *implication* problem. The satisfiability problem is to determine whether or not an input set of eCFDs (resp. CFD^{c}s, CFD^{p}s, and CIND^{p}s) makes sense, *i.e.,* whether there exists a nonempty database that satisfies the set of eCFDs (resp. CFD^{c}s, CFD^{p}s, and CIND^{p}s). And the implication problem is to determine whether or

not an eCFD (resp. CFD^c, CFD^p, and CIND^p) is entailed by a given set of eCFDs (resp. CFD^{c}s, CFD^{p}s, and CIND^{p}s) . These are important in validation and optimization of data-cleaning processes in practice.

We show that despite the increased expressive power, eCFDs (resp. CFD^{c}s, CFD^{p}s, and CIND^{p}s) do not make our lives harder. More specially, we show the following:

- for eCFDs, CFD^{c}s, and CFD^{p}s, their satisfiability problem (resp. their implication problem) remains NP-complete (resp. coNP-complete), *the same as* their CFD counterparts; and
- for CIND^{p}s, their satisfiability problem (resp. their implication problem) remains $O(1)$ time (resp. EXPTIME-complete), *the same as* their CIND counterparts.

Organizations. Sections 3.2, 3.3, and 3.4 introduce eCFDs, CFD^{c}s, and CFD^{p}s and CIND^{p}s, respectively, and establish the complexity bounds for reasoning abouteCFDs, CFD^{c}s, and CFD^{p}s and CIND^{p}s, respectively. Related work is discussed in Section 3.5.

3.2 Extending CFDs with Disjunctions and Negations

In this section, we propose an extension of conditional functional dependencies (CFDs), referred to as *extended Conditional Functional Dependencies* (eCFDs). And we study the satisfiability problem and the implication problem of eCFDs.

3.2.1 eCFDs: An Extension of CFDs

Conditional functional dependencies (CFDs) have recently been introduced in [FGJK08] for data cleaning. CFDs extend functional dependencies (FDs) by enforcing patterns of semantically related values, and have proved more effective in catching data inconsistencies than FDs, which were currently the basis of many data-cleaning tools [ABC03b, BFFR05, CM05, FPL$^+$01, Wij05].

To capture inconsistencies that commonly arise in real-life data, however, one often needs to use more expressive constraints, as illustrated by the following example.

Example 3.2.1: Let us consider a schema similar to the one used in [FGJK08]: $\mathsf{cust}(\mathsf{AC}, \mathsf{PN}, \mathsf{NM}, \mathsf{STR}, \mathsf{CT}, \mathsf{ZIP})$. It specifies a customer in New York State in terms of the customer's phone (area code (AC), phone number (PN)), name (NM), and address (street (STR), city (CT), zip code (ZIP)). An instance D_0 of cust is shown in Fig. 3.1.

One may want to specify the CFD below on cust:

ϕ_1: $(\mathsf{CT} \rightarrow \mathsf{AC},\ \{(\text{Albany} \parallel 518), (\text{Troy} \parallel 518), (\text{Colonie} \parallel 518)\})$

The CFD is a pair consisting of an embedded standard FD and a pattern tableau. It states that the FD $\mathsf{CT} \rightarrow \mathsf{AC}$ (city uniquely determines area code) holds if CT is Albany, Troy or Colonie; and in addition, the pattern tableau refines the FD by enforcing bindings between cities and area codes: if CT is one of these cities, then AC must be 518. This CFD identifies tuple t_1 in Fig. 3.1 as an error: CT is Albany but AC is not 518. This error cannot be caught by traditional FDs.

A cursory examination of New York area codes reveals that most cities in the state have a *unique* area code, except NYC and LI (Long Island). Such situations commonly arise in practice, and it is useful to capture this as constraints when checking inconsistencies of the data. Unfortunately, it cannot be defined as a standard FD or even a CFD.

This, however, can be expressed as the constraint below:

ϕ_2: $\mathsf{CT} \notin \{\text{NYC}, \text{LI}\} \rightarrow \mathsf{AC}$

which assures that the FD $\mathsf{CT} \rightarrow \mathsf{AC}$ holds if CT is *not in* the set $\{\text{NYC}, \text{LI}\}$, instead of on the entire cust database.

For NYC, one can write the following constraint:

ϕ_3: $\mathsf{CT} \in \{\text{NYC}\} \rightarrow \emptyset$ with $\mathsf{AC} \in \{212, 718, 646, 347, 917\}$

This asserts that when CT is NYC, AC must be *either* 212, 718, 646, 347, *or* 917. That is, here CT is associated with *a disjunction of options* rather than with a *single* value, and chooses one from the *multiple* choices. Again this is common in practice. With ϕ_3 we can identify tuple t_4 of Fig. 3.1 as an error: 100 is *not* one of the area codes associated with NYC. Similarly one can specify the area codes for LI.

However, these constraints cannot be defined as CFDs: they are specified with negation (ϕ_2) and disjunction (ϕ_3), beyond the expressive power of CFDs. □

eCFDs. We now define extended Conditional Functional Dependencies (eCFD), which can express all the constraints we have encountered in Example 3.2.1.

Consider a relation schema R defined over a finite set of attributes, denoted by $\mathsf{attr}(R)$. For each A in $\mathsf{attr}(R)$ we denote by $\mathsf{dom}(A)$ the domain of attribute A, which can be infinite or finite (with at least two elements).

Syntax. An eCFD φ is a triple $(R : X \rightarrow Y,\ Y_p,\ T_p)$, where (1) $X, Y, Y_p \subseteq \mathsf{attr}(R)$, and $Y \cap Y_p = \emptyset$; (2) $X \rightarrow Y$ is a standard FD, referred to as the *embedded functional dependency* of φ; and (3) T_p is a *pattern tableau* consisting of a finite number of pattern

	AC	PN	NM	STR	CT	ZIP
t_1:	718	1111111	Mike	Tree Ave.	Albany	12238
t_2:	518	2222222	Joe	Elm Str.	Colonie	12205
t_3:	518	2222222	Jim	Oak Ave.	Troy	12181
t_4:	100	1111111	Rick	8th Ave.	NYC	10001
t_5:	212	3333333	Ben	5th Ave.	NYC	10016
t_6:	646	4444444	Ian	High St.	NYC	10011

Figure 3.1: Instance D_0 of the cust relation

φ_1 = (cust: [CT] $\rightarrow$ [AC], $\emptyset$, T_1), where the pattern tableau T_1 is

CT	AC	$\emptyset$
$\overline{\{\text{NYC, LI}\}}$	_	
{Albany, Troy, Colonie}	{518}	

φ_2 = (cust: [CT] $\rightarrow \emptyset$, AC, T_2), where the pattern tableau T_2 is

CT	$\emptyset$	AC
{NYC}		{212, 718, 646, 347, 917}

Figure 3.2: Example eCFDs

tuples over the attributes in $X \cup Y \cup Y_p$, such that for any tuple $t_p \in T_p$ and for each attribute A in $X \cup Y \cup Y_p$, $t_p[A]$ is either an unnamed variable '_', a set S or a complement set $\overline{S}$, where S is a finite subset of $\mathsf{dom}(A)$. If A appears in both X and $Y \cup Y_p$, we use $t_p[A_L]$ and $t_p[A_R]$ to indicate the A field of t_p corresponding to A in X and $Y \cup Y_p$, respectively. We denote X by $\mathsf{LHS}(\varphi)$ and $Y \cup Y_p$ by $\mathsf{RHS}(\varphi)$.

Example 3.2.2: Constraints $\phi_1 - \phi_3$ of Example 3.2.1 can be expressed as the eCFDs shown in Fig. 3.2. In φ_1, $X = \{\mathsf{CT}\}$, $Y = \{\mathsf{AC}\}$ and $Y_p = \emptyset$. In φ_2, $X = \{\mathsf{CT}\}$, $Y = \emptyset$ and $Y_p = \{\mathsf{AC}\}$. Here φ_1 expresses ϕ_1 and ϕ_2, while φ_2 represents ϕ_3. We use $\|$ to separate X, Y and Y_p. □

Observe that *each pattern tuple* is actually a constraint that enforces binding of semantically related values, and is referred to as a *pattern* constraint in the eCFD.

Semantics. Let us consider $Z \subseteq X \cup Y \cup Y_p$, a tuple t in an instance I of R, and a *pattern tuple* $t_p \in T_p$. We say that the *data* tuple $t[Z]$ *matches* the pattern tuple $t_p[Z]$, denoted by $t[Z] \asymp t_p[Z]$, if for each $A \in Z$, (1) if $t_p[A] =$ '_', then $t[A] \in \mathsf{dom}(A)$, *i.e.,* an arbitrary value; (2) if $t_p[A] = S$, then $t[A] \in S$; and (3) if $t_p[A] = \overline{S}$, then $t[A] \notin S$.

For example, consider t_1, t_4 of Fig. 3.1 and the first pattern tuple t_p of φ_1 in Fig. 3.2.

Then $t_1[\mathsf{CT},\mathsf{AC}] \asymp t_p[\mathsf{CT},\mathsf{AC}]$ since $t_1[\mathsf{CT}] \notin \{\text{NYC, LI}\}$, and $t_1[\mathsf{AC}] \asymp$ '_'. However, $t_4[\mathsf{CT},\mathsf{AC}] \not\asymp t_p[\mathsf{CT},\mathsf{AC}]$ since $t_4[\mathsf{CT}] \in \{\text{NYC, LI}\}$.

A relation I of R *satisfies* eCFD φ, denoted by $I \models \varphi$, if for *each* pattern tuple $t_p \in T_p$, the following holds. Let $I(t_p) = \{t \in I \mid t[X] \asymp t_p[X]\}$, which is the set of tuples t in I such that $t[X]$ matches $t_p[X]$. Then (1) $I(t_p)$ must satisfy the embedded FD $X \to Y$, *i.e.,* for any two tuples t_1, t_2 in $I(t_p)$, if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$; and furthermore, (2) for *each* tuple $t \in I(t_p)$, $t[Y,Y_p] \asymp t_p[Y,Y_p]$, *i.e.,* all tuples t in $I(t_p)$ must match the pattern $t_p[Y,Y_p]$.

Intuitively, $I(t_p)$ identifies the set of tuples on which the constraint t_p is defined, *i.e.,* the constraint only applies to the tuples in I that match the pattern $t_p[X]$. Both, the embedded FD $X \to Y$ and the pattern $t_p[Y,Y_p]$, are enforced on the tuples in $I(t_p)$.

Example 3.2.3: Consider the database D_0 of Fig. 3.1 and the first pattern tuple t_p in φ_1. Here $D_0(t_p) = \{t_1, t_2, t_3\}$, *i.e.,* the tuples whose CT attribute is neither NYC nor LI. In other words, the constraint specified by t_p does *not* apply to the *entire* D_0; it holds *conditionally* on D_0, *i.e.,* only on $D_0(t_p)$.

The database D_0 satisfies neither φ_1 nor φ_2. Even though $t_1[\mathsf{CT}] \asymp t'_p[\mathsf{CT}]$ and t_1 does not violate the FD $\mathsf{CT} \to \mathsf{AC}$, where t'_p is the second pattern tuple of φ_1, t_1 violates φ_1 since $t_1[\mathsf{AC}] \not\asymp t'_p[\mathsf{AC}]$. The tuple t_4 violates φ_2 since $t_1[\mathsf{AC}] \not\asymp t''_p[\mathsf{AC}]$ although $t_1[\mathsf{CT}] \asymp t''_p[\mathsf{CT}]$, where t''_p is the pattern tuple of φ_2 (here CT is the Y_p attribute of φ_2). These tell us that a *single* tuple may violate an eCFD while it takes two tuples to violate a standard FD. □

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

Remarks. (1) eCFDs support *negations* ($\overline{S}$, *e.g.,* the first pattern tuple of φ_1) and *disjunctions* (S, *e.g.,* φ_2 in which the area code for NYC is specified as *either* 212, 718, 646, 347 *or* 917). (2) Conditional functional dependencies (CFDs) introduced in [FGJK08] are a special case of eCFDs. Recall that a CFD is of the form $(R : X \to Y,\ T_p)$, in which each pattern tuple consists of either '_' or a *single* constant value. Hence, a CFD can be written as an eCFD $\varphi = (R : X \to Y,\ \emptyset,\ T'_p)$, where T'_p is identical to T_p except that each constant a in T_p is replaced with $\{a\}$ in T'_p. That is, a CFD is an eCFD with neither negation nor disjunction. Since CFDs extend standard FDs, so do eCFDs.

3.2.2 Reasoning about eCFDs

In this section we investigate the satisfiability and implication analysis of eCFDs. These are classical decision problems associated with any constraint language.

The *satisfiability problem* for eCFDs is to decide, given a set Σ of eCFDs on a relation schema R, whether or not there exists a nonempty instance I of R such that $I \models \Sigma$.

The *implication problem* for eCFDs is to determine, given a set Σ of eCFDs and another eCFD φ defined on the same relation schema R, whether or not $\Sigma \models \varphi$, *i.e.,* whether for every instance I of R, if $I \models \Sigma$ then also $I \models \varphi$.

The main result of this section is that despite the increased expressive power of eCFDs, they retain the same complexity bounds for these static analysis as CFDs.

Satisfiability. As shown in [FGJK08], CFDs may not be satisfiable. It is thus not surprising that the same holds for eCFDs.

Example 3.2.4: Consider an eCFD ψ_3 on cust databases: (cust: [CT] $\rightarrow$ [CT], $\emptyset$, $\{(\overline{\{\text{NYC}\}} \parallel \{\text{NYC}\} \parallel \emptyset), (\{\text{NYC}\} \parallel \{\text{LI}\} \parallel \emptyset)\}$). This eCFD is not satisfiable. Indeed, for any cust instance I and any tuple t in I, if $t[\text{CT}] = \text{NYC}$, then ψ_3 requires it to be LI; but ψ_3 forces it to be NYC again. □

This highlights the need for the satisfiability analysis of eCFDs: it is necessary to determine whether or not the given eCFDs are not dirty themselves before one uses the eCFDs to detect inconsistencies in a database, which is typically much larger than the set of constraints.

It is known that the satisfiability problem for CFDs is NP-complete [FGJK08]. The result below shows that eCFDs do not make the satisfiability analysis more complicated.

Theorem 3.2.1: *The satisfiability problem for* eCFD*s is* NP*-complete.* □

Proof: Since CFDs are a special case of eCFDs, the NP-hardness of the problem follows immediately from the NP-hardness of the satisfiability problem for CFDs [FGJK08].

We then show that the problem is in NP by giving an NP algorithm for checking the satisfiability for eCFD. The key idea is the following *small model property*: if a given set Σ of eCFDs on a relational schema R is satisfiable, then there exists an R instance I consisting of a *single* tuple t that satisfies Σ, *i.e., $I \models \Sigma$*.

Assume that the relational schema R has attributes $\mathsf{attr}(R) = \{A_1, \ldots, A_n\}$. Now, for $i \in [1,n]$, let $\mathsf{adom}(A_i)$ consist of the union of the data values in all sets S defined on the A_i-attribute appearing in the pattern tuples of all eCFDs in Σ. Here the sets S can appear either positively, *i.e.,* as the set S itself, or negatively, *i.e.,* as its complement $\bar{S}$. Moreover, for each $i \in [1,n]$, we add an extra distinct value – not used anywhere –

from $\mathsf{dom}(A_i)$, if it exists. Note that the size of the domain $\mathsf{adom}(A_i)$ ($i \in [1,n]$) is a polynomial in the size of Σ.

Then, it is easy to verify that if $I = \{t\}$ and $I \models \Sigma$, then there exists a mapping ρ from $t[A_i]$ to $\mathsf{adom}(A_i)$ such that $I_t = \{\rho(t)\} = \{(\rho(t[A_1]), \ldots, \rho(t[A_n]))\}$ and $I_t \models \Sigma$. That is, if Σ is satisfiable, there must exist a single-tuple instance satisfying Σ, and for each attribute A_i ($i \in [1,n]$), the tuple on the attribute A_i only draws values from $\mathsf{adom}(A_i)$.

We are now ready to give the details of the NP algorithm, as below:

(a) Guess a single tuple t of R such that $t[A_i] \in \mathsf{adom}(A_i)$.

(b) Check whether $I = \{t\}$ satisfies Σ, and return 'yes' if $I \models \Sigma$.

Both steps (a) and (b) can be done in PTIME, and it is easy to verify the correctness of the NP algorithm. Thus, we conclude that the satisfiability problem for eCFDs is indeed NP-complete. □

Implication. Due to the presence of pattern tuples in eCFDs, one expects the number of eCFDs to be larger than their FD counterparts. A natural optimization strategy for cleaning data with eCFDs is by removing redundancies in a given set of eCFDs, *i.e.,* by removing eCFDs and pattern tuples that are entailed by other eCFDs. This calls for the implication analysis of eCFDs.

The implication problem is coNP-complete for CFDs [FGJK08]. The complexity remains unchanged for eCFDs:

Theorem 3.2.2:. *The implication problem for e*CFD*s is co*NP*-complete.* □

Proof: Since CFDs are a special case of eCFDs, the coNP-hardness of the problem follows immediately from the coNP-hardness of the implication problem for CFDs [FGJK08].

We then show that the problem is in coNP by giving an NP algorithm for its complement. That is, given a set $\Sigma \cup \{\varphi\}$ of eCFDs on a relational schema R, check whether or not $\Sigma \not\models \varphi$.

Similar to the NP algorithm in the proof of Proposition 3.2.1, the NP algorithm again is based on a *small model property*. However, we need *two* tuples instead of *one* tuple. More precisely, it is easily verified that if there exists an instance $I \models (\Sigma \cup \{\neg\varphi\})$, then there exists a two-tuple subset $\{s,t\} \subseteq I$ that satisfies Σ but does not satisfy φ. It is essential to use two tuples here because some φ's can only be violated by means of two tuples.

Therefore, it is sufficient to consider two-tuple instances I. For any attribute $A_i \in$

$\mathsf{attr}(R)$, denote by $\mathsf{adom}(A_i)$ the set of constants appearing in any pattern tuple and any eCFD, plus at most two extra distinct constant from the domain of A_i, if these exist. Again, the sizes of the domain $\mathsf{adom}(A_i)$ ($i \in [1,n]$) is a polynomial in the size of Σ

It is easy to verify that if $I = \{t_1, t_2\}$ and $I \models \Sigma$ but $I \not\models \varphi$, then there exists a mapping ρ from $t_1[A_i]$ and $t_2[A_i]$ to $\mathsf{adom}(A_i)$ such that $I' = \{\rho(t_1), \rho(t_2)\}$ and $I' \models \Sigma$, but not $I' \models \varphi$. That is, if $\Sigma \not\models \varphi$, there must exist a two-tuple instance $I = \{s,t\}$ such that $I \models \Sigma$, $I \not\models \varphi$ and $s[A_i], t[A_i] \in \mathsf{adom}(A_i)$ for each $i \in [1,n]$.

The NP-algorithm works as follows:

(a) Guess two tuples s and t such that for each attribute A_i ($i \in [1,n]$), both $s[A_i]$ and $t[A_i]$ take their values from $\mathsf{adom}(A_i)$.

(b) Check whether $I = \{s,t\} \models \Sigma$ and $I \not\models \varphi$, and return 'yes' if so.

Both steps (a) and (b) can be done in PTIME, and it is easy to verify the correctness of the NP algorithm. Thus, we conclude that the implication problem for eCFDs is coNP-complete. □

Special case. A tractable special case is identified in [FGJK08]: if the given CFDs involve no attributes that have a *finite* domain, then the satisfiability and implication analysis are in PTIME. This is no longer the case for eCFDs, since we can enforce, via eCFDs, an attribute A to draw values from a finite set only, no matter whether $\mathsf{dom}(A)$ is infinite or not.

Theorem 3.2.3: *Both the satisfiability problem and the implication problem for e*CFD*s remain* NP*-complete and co*NP*-complete, respectively, in the absence of finite-domain attributes.* □

Proof: Following from Propositions 3.2.1 and 3.2.2, it suffices to show that the satisfiability problem and the implication problem for eCFDs are NP-hard and coNP-hard, respectively, in the absence of finite-domain attributes.

(1) For the satisfiability problem, we show the NP-hardness by reduction from the satisfiability problem for CFDs with finite-domain attributes, which is NP-complete[FGJK08].

Given a set Σ of CFDs on a relational schema $R = (A_1, \ldots, A_n)$, we define another relational schema $R' = (A'_1, \ldots, A'_n)$ with only infinite-domain attributes. We define a set Σ' of eCFDs as below.

- For each CFD φ, we include an eCFD φ' in Σ' by renaming all attributes A to A', and changing the constants a in the pattern tuples of φ to a set format $\{a\}$, as the pattern tuples of φ'.

- For each finite-domain attribute A_i ($1 \leq i \leq n$) with a finite domain $\mathsf{dom}(A_i)$, we include an eCFD $\varphi_{A_i} = (R' : [A'_i] \rightarrow \emptyset, A'_i, \{(\_ \parallel \emptyset \parallel \mathsf{dom}(A_i))\})$ in Σ'. The eCFD ψ_{A_i} forces R' tuples on the attribute A'_i to take values only from the finite domain $\mathsf{dom}(A_i)$.

Clearly, Σ' is satisfiable if and only if Σ is satisfiable.

(2) Similarly, for the implication problem, we can show the coNP-hardness by reduction from the implication problem for CFDs with finite-domain attributes, which is coNP-complete[FGJK08]. □

3.3 Extending CFDs with Cardinality Constraints and Synonym Rules

In this section, we propose an extension of conditional functional dependencies (CFDs), denoted by CFD^{c}s, to express cardinality constraints, domain-specific conventions, and patterns of semantically related constants in a uniform constraint formalism.And we study the satisfiability problem and the implication problem of CFD^{c}s.

3.3.1 CFD^{c}s: An Extension of CFDs

Conditional functional dependencies (CFDs) have recently been studied for detecting inconsistencies in relational data [FGJK08]. These dependencies are an extension of functional dependencies (FDs) by enforcing patterns of semantically related data values. In contrast to traditional FDs that were developed for improving the quality of schema, CFDs aim to improve the quality of the data. That is, CFDs are to be used as data-quality rules such that errors and inconsistencies in the data can be detected as violations of these dependencies.

While CFDs are capable of capturing more errors than traditional FDs, they are not powerful enough to detect certain inconsistencies commonly found in real-life data. To illustrate this, let us consider an example.

Example 3.3.1: Consider a relation schema:

sale(FN: string, LN: string, street: string, city: string, state: string, country: string, zip: string, item: string, type: string)

where each tuple specifies an item of a certain type purchased by a customer. Each customer is specified by her name (FN, LN) and address (street, city, state, country, zip). An instance D_0 of the sale schema is shown in Fig. 3.3.

	FN	LN	street	city	state	country	zip	item	type
t_1:	Joe	Brady	Mayfield	EDI	N/A	UK	EH4 8LE	CD1	regular
t_2:	Mark	Webber	Crichton	EDI	NY	United Kingdom	EH4 8LE	CD2	sale
t_3:	John	Hull	Queen	EDI	N/A	UK	EH4 8LE	CD3	regular
t_4:	William	Smith	5th Ave	NYC	NY	US	10016	book1	sale
t_5:	Bill	Smith	5th Ave	NYC	NY	US	10016	book2	sale
t_6:	Bill	Smith	5th Ave	NYC	NY	US	10016	book3	sale

Figure 3.3: An instance of the sale relation schema

CFDs on sale data include the following:

ϕ_1: ([country, zip] $\rightarrow$ street, t_p^1), and t_p^1 = (UK, _ || _))

ϕ_2: (country $\rightarrow$ state, t_p^2), where t_p^2 = (UK || N/A)

Here ϕ_1 asserts that for customers in the UK, zip code uniquely determines street. It uses a tuple t_p^1 to specify a pattern: country = UK, zip = '_' and street = '_', where '_' can take an arbitrary value. It is an "FD" that is to hold on the subset of tuples that satisfies the pattern, *e.g.,* $\{t_1, t_3\}$ in D_0, rather than on the entire D_0 (in the US, for example, zip does not determine street). It is not a traditional FD since it is defined with constants. Similarly, ϕ_2 assures that for any address in the UK, state must be N/A (non-applicable); this is enforced by pattern tuple t_p^2: country = UK and state = N/A.

When these CFDs are used as data quality rules, one can see that either t_1 or t_3 is "dirty": they violate the rule ϕ_1. Indeed, t_1 and t_3 are about customers in the UK and they have the same zip; however, they have different streets.

A closer examination of D_0 reveals that tuple t_2 is not error-free either. Indeed, t_2 is about a transaction for a UK customer, but (a) its state is NY rather than N/A, and (b) while its zip is the same as that of t_1 and t_3, it has a street not found in t_1 or t_3. However, these violations cannot be detected by ϕ_1 and ϕ_2. Indeed, these CFDs are specified with the pattern country = UK, and do not apply to tuples with country = "United Kingdom". Although UK and United Kingdom refer to the same country, they are not treated as equal by the equality operator adopted by CFDs and FDs. In other words, CFDs and FDs do not observe domain-specific abbreviations and conventions.

Another issue concerns *cardinality constraints* commonly found in practice, which require that the number of tuples with a certain pattern does not exceed a predefined bound. An example is that each customer is allowed to purchase at most two distinct items on sale (with type = sale). As another example, on a school database, one may want to specify that a CS student can register for at most six courses each semester. These constraints can be expressed as neither FDs nor CFDs. □

These practical concerns highlight the following questions. Can one extend CFDs to express cardinality constraints and synonym rules (domains-specific abbreviations and conventions)? Can we find an extension such that it does not increase the complexity for reasoning about these dependencies? Indeed, we want a balance between the expressive power needed to deal with these issues, and the complexity for static analyses of the dependencies.

CFD^{c}s. We now define an extension of CFDs, which can express all the constraints we have encountered in Example 3.3.1.

Consider a relation schema R defined over a set of attributes, denoted by $\mathsf{attr}(R)$. For each attribute $A \in \mathsf{attr}(R)$, its domain is specified in R, denoted as $\mathsf{dom}(A)$. As will be seen in Sections 3.3.2 and 3.3.3, the domains of attributes have substantial impact on the complexity of satisfiability and implication analyses of CFD^{c}s.

Syntax. A CFD^c φ defined on schema R is a triple $R(X \to Y, t_p, c)$, where (1) $X \to Y$ is a standard FD, referred to as the FD *embedded in* φ; (2) t_p is a tuple with attributes in X and Y, referred to as the *pattern tuple* of φ, where for each A in $X \cup Y$, $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed (yet marked) variable '_' that draws values from $\mathsf{dom}(A)$; and (3) c is a positive integer. We refer to φ also as a *conditional functional dependency*.

Intuitively, t_p specifies a pattern of semantically related values for X and Y attributes: for any tuple t in an instance of R, if $t[X]$ has the pattern $t_p[X]$, then $t[Y]$ must observe the pattern $t_p[Y]$. Furthermore, for all those tuples t such that $t[X]$ has pattern $t_p[X]$, if we group $t[Y]$ values by $t[X]$, then the number of distinct values in (*i.e.,* the cardinality of) each group is not allowed to exceed the bound c. In particular, when $c = 1$, $t[X]$ uniquely determines $t[Y]$, *i.e.,* the FD embedded in φ is enforced on those tuples having a $t_p[X]$ pattern.

If A occurs in both X and Y, we use $t_p[A_L]$ and $t_p[A_R]$ to indicate its occurrence in X and Y, respectively. We separate the X and Y attributes in t_p with '$\|$', and denote X as $\mathsf{LHS}(\varphi)$ and Y as $\mathsf{RHS}(\varphi)$. We write φ as $(X \to Y,\ t_p, c)$ when R is clear from the context.

Example 3.3.2: CFDs ϕ_1 and ϕ_2 of Example 3.3.1 can be expressed as CFD^{c}s below, in which t_p^1 and t_p^2 are pattern tuples given in Example 3.3.1:

φ_1: $([\mathsf{country}, \mathsf{zip}] \to \mathsf{street},\ t_p^1,\ 1)$,

φ_2: $(\mathsf{country} \to \mathsf{state},\ t_p^2,\ 1)$.

The cardinality constraint described in Example 3.3.1 can also be written as a CFD^c φ_3: $(\mathsf{fd}, t_p^3, 2)$, where FD fd and pattern tuple t_p^3 are:

fd: FN, LN, street, city, state, country, zip, type $\rightarrow$ item,
t_p^3 = (_, _, _, _, _, _, _, sale $\|$ _),

assuring that no customers may buy more than two distinct items with type = sale. □

Semantics of CFD^c**s.** To give the semantics of CFD^{c}s, we first extend the equality relation and revise the match operator of [FGJK08].

An extension of equality. We use a finite binary relation R_c to capture synonym rules. For values a and b, $R_c(a,b)$ indicates that a and b refer to the same real-world entity. For example, R_c("William", "Bill") and R_c("United Kingdom", "UK"). We assume *w.l.o.g.* that R_c is symmetric: if $R_c(a,b)$ then $R_c(b,a)$. However, R_c may *not* be transitive: from R_c("New York State", "NY") and R_c("NY", "New York City") it does not follow that R_c("New York State", "New York City").

In the sequel we assume that R_c is predefined, as commonly found in practice.

We define a binary operator $\doteq$ on constants such that for any values a and b, $a \doteq b$ iff (1) $R_c(a,b)$ or $a = b$, (2) $b \doteq a$, or (3) there exists a value c such that $a \doteq c$ and $b \doteq c$. For example, "United Kingdom" $\doteq$ "UK".

The operator $\doteq$ naturally extends to tuples: $(a_1,\ldots,a_k) \doteq (b_1,\ldots,b_k)$ iff for all $i \in [1,k]$, $a_i \doteq b_i$. Observe that given a fixed R_c, whether $a \doteq b$ can be decided in polynomial time.

Matching operator. We revise the binary operator $\asymp$ of [FGJK08] defined on constants and '_' as follows: $\eta_1 \asymp \eta_2$ if either (a) η_1 and η_2 are constants and $\eta_1 \doteq \eta_2$, or (b) one of η_1, η_2 is '_'. The operator $\asymp$ extends to tuples, *e.g.,* $(a,b) \asymp (\_,b)$ but $(a,b) \not\asymp (\_,c)$ if $b \not\doteq c$.

Semantics. Based on the operators $\doteq$ and $\asymp$, we now give the semantics of CFD^c $\varphi = R(X \rightarrow Y, t_p, c)$.

An instance D of schema R *satisfies* φ, denoted by $D \models \varphi$, iff for each tuple t in D, if $t[X] \asymp t_p[X]$, then (1) $t[Y] \asymp t_p[Y]$, and (2) $|\pi_Y(\sigma_{X \doteq t[X]} D)| \leq c$, *i.e.,* for all tuples t' in D such that $t'[X] \doteq t[X]$, there exist at most c distinct $t'[Y]$ values. Here π and σ are the projection and selection operators in relational algebra, respectively; and $|S|$ denotes the cardinality of a set S in which no two elements a, b are comparable by $a \doteq b$.

Intuitively, φ is a constraint defined on the set of tuples $D_\varphi = \{t \mid t \in D, t[X] \asymp t_p[X]\}$ such that (a) for each $t \in D_\varphi$, the pattern $t_p[Y]$ is enforced on $t[Y]$; (b) for each set of tuples in D_φ grouped by X attribute values, the number of their distinct Y values is bounded by the constant c; that is, φ expresses a *cardinality constraint* on the Y values of those tuples grouped by X; and (c) synonym rules are captured by the extension

$\doteq$ of the equality relation. Note that φ is defined on the subset D_φ of D identified by $t_p[X]$, rather than on the entire D.

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

Example 3.3.3: Assume that R_c consists of ("United Kingdom", "UK") and ("William", "Bill"). Recall instance D_0 of Fig. 3.3 and CFD^{c}s φ_1, φ_2 and φ_3 of Example 3.3.2. Observe the following: (a) tuple t_2 in D_0 violates φ_2, since $t_2[\mathsf{country}] \asymp$ UK but $t_2[\mathsf{state}] \not\asymp$ N/A; (b) t_1, t_2 and t_3 violate φ_1 since they are UK records with the same zip code, but they have different streets; (c) t_4, t_5 and t_6 violate φ_3, since they agree on name and address (note that William $\doteq$ Bill), all have type = sale, but they have three distinct items, their item attributes have three distinct values, beyond the bound 2. □

Three special cases of CFD^{c}s are worth mentioning. (a) Traditional FDs are CFD^{c}s in which c is 1 and the pattern tuple consists of '_' only. (b) CFDs of [FGJK08] are CFD^{c}s in which c is fixed to be 1. (c) *Constant* CFD^c*s* are CFD^{c}s in which the pattern tuples consist of constants only, *i.e.,* they do not contain '_'.

3.3.2 The Satisfiability Analysis

A central technical problem associated with CFD^{c}s is the satisfiability problem.

The *satisfiability problem* for CFD^{c}s is to determine, given a set Σ of CFD^{c}s on a schema R, whether or not there exists a nonempty instance D of R such that $D \models \Sigma$. The set Σ is said to be *satisfiable* if such an instance exists.

Intuitively, the satisfiability problem is to decide whether a set of CFD^{c}s makes sense or not. When CFD^{c}s are used as data quality rules, the satisfiability analysis helps us detect whether the rules are dirty themselves.

Any set of FDs is satisfied by a nonempty relation. In contrast, the satisfiability problem becomes NP-complete for CFDs [FGJK08]. Since CFD^{c}s subsume CFDs, the satisfiability problem for CFD^{c}s is at least as hard as for CFDs.

Example 3.3.4: Consider a schema $R(A, B, C)$, and a set Σ_1 consisting of three CFD^{c}s defined on R: $\psi_1 = (A \to B,\ (\mathsf{true} \parallel b),\ 1)$, $\psi_2 = (A \to B,\ (\mathsf{false} \parallel b),\ 1)$, and $\psi_3 = (C \to B,\ (\_ \parallel b'),\ 1)$, where $\mathsf{dom}(A)$ is Boolean, and $b \neq b'$. Then Σ_1 is not satisfiable. Indeed, for any nonempty instance D of R and any tuple t in D, ψ_3 requires $t[B]$ to be b' no matter what value $t[C]$ is, whereas ψ_1 and ψ_2 force $t[B]$ to be b no matter whether $t[A]$ is true or false. □

The intractability. Despite the increased expressive power, CFD^{c}s do not complicate the satisfiability analysis. Indeed, the satisfiability problem for CFD^{c}s remains in NP. The proof for the result below is an extension of Theorem 3.2 in [FGJK08], its counterpart for CFDs.

Theorem 3.3.1: *The satisfiability problem for* CFD^c*s is* NP*-complete.* □

Proof: It is known that the satisfiability problem is already NP-hard even for constant CFDs [FGJK08]. Since CFD^{c}s subsume CFDs, the NP lower bound for CFDs carries over to CFD^{c}s.

We show the upper bound by presenting an NP algorithm that, given a set Σ of CFD^{c}s on a schema R, checks whether Σ is satisfiable. Similar to CFDs [FGJK08], CFD^{c}s have a *small model property*: if there is a nonempty instance D of R such that $D \models \Sigma$, then for any $t \in D$, $\{t\}$ is an instance of R and $\{t\} \models \Sigma$. Thus it suffices to consider single-tuple instances $\{t\}$ for deciding whether Σ is satisfiable.

Assume *w.l.o.g.* that $\mathsf{attr}(R) = \{A_1, \ldots, A_n\}$. For each $i \in [1,n]$, define the active domain of A_i to be a set $\mathsf{adom}(A_i)$ consisting of all constants of $t_p[A_i]$ for all pattern tuples t_p in Σ, plus an extra distinct value in $\mathsf{dom}(A_i)$ (if there exists one). Then it is easy to verify that Σ is satisfiable iff there exists a mapping ρ that assigns a value in $\mathsf{adom}(A_i)$ to $t[A_i]$ for each $i \in [1,n]$ such that $D = \{(\rho(t[A_1]), \ldots, \rho(t[A_n]))\}$ and $D \models \Sigma$.

Based on these, we give the NP algorithm as follows: (a) Guess a single tuple t of R such that $t[A_i] \in \mathsf{adom}(A_i)$ for each $i \in [1,n]$. (b) Check whether $\{t\} \models \Sigma$. If so it returns "yes", and otherwise it repeats steps (a) and (b). Note that step (b) involves checking whether $x \doteq y$, which can be done in PTIME in the sizes of Σ and R_c, where R_c is the relation given in the definition of $\doteq$. Hence the algorithm is in NP, and so is the satisfiability problem. □

A tractable case. As shown by Example 3.3.4, the complexity is introduced by attributes in CFD^{c}s with a finite domain. This motivates us to consider the following special case.

A set Σ of CFD^c is said to be *bounded by a constant* k if at most k attributes in the CFD^{c}s of Σ have a finite domain. In particular, when $k = 0$, all CFD^{c}s in Σ are defined in terms of attributes with an infinite domain.

Bounded CFD^{c}s make our lives much easier. Indeed, an extension of the proof of Proposition 3.5 in [FGJK08] suffices to show the following.

Proposition 3.3.2: *It is in* PTIME *to determine whether a set* Σ *of* CFD^c*s is satisfiable if* Σ *is bounded by a constant* k*.* □

Proof: When Σ is bounded by k, we develop a PTIME algorithm to determine whether Σ is satisfiable, which is based on a modified chase (see, *e.g.,* [AHV95] for the chase), and the small model property identified in the proof of Theorem 3.3.1. The algorithm is an extension of the one for CFDs (Proposition 3.5 in [FGJK08]) to further deal with finite domain attributes and the $\doteq$ operator. Assume *w.l.o.g.* that Σ is defined on a schema R, and only attributes A_i in CFD^{c}s of Σ have a finite domain, for $i \in [1,k]$.

The algorithm checks whether there exists a tuple t of R such that $t \models \Sigma$. Initially $t[A]$ is a distinct variable x_A for each $A \in \mathsf{attr}(R)$. For all $i \in [1,k]$ and for each value in $\mathsf{dom}(A_i)$ assigned to x_{A_i}, the algorithm does the following.

(a) For each CFD^c $\phi = R(X \to Y, t_p, c)$ in Σ, chase t using ϕ: if $t[X] \asymp t_p[X]$, then change $t[Y]$ such that $t[Y] \asymp t_p[Y]$ as long as $t[Y]$ does not already contain a constant that does not match the corresponding field in $t_p[Y]$.

Here we extend the match operator $\asymp$ to accommodate variables x_B: $x_B \asymp \_$, but $x_B \not\asymp \eta$ when η is a constant or a variable.

(b) For each attribute $B \in \mathsf{attr}(R)$, if $t[B]$ is still x_B after step (a), assign a distinct value from $\mathsf{dom}(B)$ to x_B, which does not appear in Σ and R_c; note that $\mathsf{dom}(B)$ must be infinite in this case by the definition of t.

(c) If $t \models \Sigma$ then return "yes"; "no" is returned if for all possible valuations to x_{A_i} for $i \in [1,k]$, it cannot instantiate t such that $t \models \Sigma$.

The algorithm is in $O(|\Sigma|^2\ |R_c|\ m^k)$ time, *i.e.,* in PTIME when k is fixed, where $|\Sigma|$ is the size of Σ, $|R_c|$ is the size of R_c (in the definition of $\doteq$), and m is the maximum cardinality of finite domains $\mathsf{adom}(A_i)$ for $i \in [1,k]$.

We next show that the algorithm returns "yes" if and only if Σ is satisfiable.

If the algorithm returns "yes", there exists a tuple t such that $t \models \Sigma$. Thus Σ is satisfiable.

Conversely, if Σ is satisfiable, there exists a tuple t such that $t \models \Sigma$. We show that the algorithm returns "yes". Initialize a tuple t' such that $t'[A_i] = t[A_i]$ for $i \in [1,k]$, and $t'[A] = x_A$ for the rest of attributes $A \in \mathsf{attr}(R)$. After step (a), for each attribute $A \in \mathsf{attr}(R)$, if $t'[A]$ is a constant, then $t'[A] \doteq t[A]$. Moreover, there exist no conflicts since $t \models \Sigma$. The assignments at step (b) are irrelevant since $t'[B]$'s instantiated at that step are not constrained by pattern tuples in Σ, and thus have no impact on whether $\{t'\}$ satisfies Σ. Thus after step (b), $\{t'\} \models \Sigma$, and the algorithm returns "yes". □

3.3.3 The Implication Analysis

We next investigate another central technical problem associated with CFD^{c}s.

Consider a set Σ of CFD^{c}s and a single CFD^c defined on the same schema R. We say that Σ *implies* φ, denoted by $\Sigma \models \varphi$, iff for all instances D of R, if $D \models \Sigma$ then $D \models \varphi$. We consider *w.l.o.g.* satisfiable Σ only.

The *implication problem* for CFD^{c}s is to determine, given a set Σ of CFD^{c}s and a CFD^c defined on the same schema, whether $\Sigma \models \varphi$.

The implication analysis helps us identify and eliminate redundant data quality rules.

As examples of the implication analysis, we present two simple results.

Proposition 3.3.3: *For any* CFD^c*s of the form:* φ*:* $R(X \to Y, t_p, c)$, φ'*:* $R(X \to Y, t_p, c')$,

(a) $\varphi \models \varphi'$ *if* $c \leq c'$*; and*

(b) if φ *is a constant* CFD^c, $\varphi \models \varphi'$ *even when* $c' = 1$ *and* $c > c'$. □

Proof: (a) This can be easily verified by the definition of CFD^{c}s. (b) We show that for any instance D of R, if $D \models \varphi$ then $D \models \varphi'$. Observe that for any tuple $t \in D$, if $t[X] \doteq t_p[X]$, then $t[Y] \doteq t_p[Y]$. Hence for all tuples t' in D, if $t'[X] \doteq t[X]$, then $t'[Y] \doteq t_p[Y]$, *i.e.,* $|\pi_Y(\sigma_{X \doteq t[X]} D)| \leq 1$. Thus $D \models \varphi'$. □

The intractability. We know that the implication problem for CFDs is coNP-complete [FGJK08]. Below we show that the upper bound remains intact for CFD^{c}s, along the same lines as its CFD counterpart (Theorem 4.3 in [FGJK08]).

In the rest of the section we consider a set Σ of CFD^{c}s and a CFD^c $\varphi = R(X \to Y, t_p, c)$ such that c is bounded by a polynomial in the sizes of Σ and φ. This assumption is acceptable since in practice, c is typically fairly small.

Theorem 3.3.4: *The implication problem for* CFD^c*s is co*NP*-complete.* □

Proof: The implication problem for constant CFDs is coNP-hard [FGJK08]. The lower bound carries over to CFD^{c}s, which subsume CFDs.

We show that the problem is in coNP by presenting an NP algorithm for its complement, *i.e.,* for deciding whether $\Sigma \not\models \varphi$. The algorithm is based on a small model property: if $\varphi = R(X \to Y, t_p, c)$ and $\Sigma \not\models \varphi$, then there exists an instance D of R with at most $c+1$ tuples such that $D \models \Sigma$ and $D \not\models \varphi$. That is, D consists of $c+1$ tuples $t_1, \ldots, t_{c+1}$ such that for all $i, j \in [1, c+1]$, $t_i[X] \asymp t_p[X]$ and $t_i[X] \doteq t_j[X]$, but either there exists $l \in [1, c+1]$ such that $t_l[Y] \not\asymp t_p[Y]$, or for all $i \neq j$, $t_i[Y] \not\doteq t_j[Y]$. Thus it suffices to consider instances D with $c+1$ tuples for deciding whether $\Sigma \not\models \varphi$.

Assume that $\mathsf{attr}(R) = \{A_1, \ldots, A_n\}$. For each $i \in [1, n]$, let $\mathsf{adom}(A_i)$ be a set consisting of (a) all constants of $t_p[A_i]$ for all pattern tuples t_p in $\Sigma \cup \{\varphi\}$, and (b) $c+1$ extra distinct values in $\mathsf{dom}(A_i)$ if they exist; if $\mathsf{dom}(A_i)$ is finite and does not have

$c+1$ extra values, let $\mathsf{adom}(A_i)$ be $\mathsf{dom}(A_i)$. Then one can verify that $\Sigma \not\models \varphi$ iff there exist mappings $\rho_1, \ldots, \rho_{c+1}$ such that ρ_i maps $t[A_j]$ to a value in $\mathsf{adom}(A_j)$ for each $j \in [1,n]$, $D = \{(\rho_1(t[A_1]),\ldots,\rho_1(t[A_n])), \ldots, (\rho_{c+1}(t[A_1]), \ldots, \rho_{c+1}(t[A_n]))\}$, $D \models \Sigma$ and $D \not\models \varphi$.

Based on these, we give the NP algorithm as follows: (a) Guess $c+1$ tuples $t_1,\ldots,t_{c+1}$ of R such that $t_j[A_i] \in \mathsf{adom}(A_i)$ for each $i \in [1,n]$ and $j \in [1,c+1]$. (b) Check whether $\{t_1,\ldots,t_{c+1}\}$ satisfies Σ, but not φ. If so the algorithm returns "yes", and otherwise it repeats steps (a) and (b). As argued in the proof of Theorem 3.3.1, step (b) can be done in PTIME in the sizes of Σ, φ and R_c. Furthermore, c is bounded by a polynomial by assumption. As a result, the algorithm is in NP and thus the implication problem is in coNP. □

Special cases. Proposition 3.3.2 shows that for a set of CFD^{c}s bounded by a constant k, the satisfiability analysis is in PTIME. This is no longer the case for the implication problem.

Theorem 3.3.5: *It is co*NP*-complete to decide, given* CFD^c*s* Σ *and* φ*, whether* $\Sigma \models \varphi$ *even when* $\Sigma \cup \{\varphi\}$ *is bounded by a constant* $k = 3$. □

Proof: The problem is in coNP by Theorem 3.3.4. We show that it is coNP-hard by reduction from 3SAT to the complement of the problem (*i.e.,* to decide whether $\Sigma \not\models \varphi$), where 3SAT is NP-complete (cf. [GJ79]). Consider an instance $\phi = C_1 \wedge \cdots \wedge C_n$ of 3SAT, where all the variables in ϕ are $x_1,\ldots,x_m$, C_j is of the form $y_{j_1} \vee y_{j_2} \vee y_{j_3}$, and moreover, for $i \in [1,3]$, y_{j_i} is either $x_{p_{ji}}$ or $\overline{x_{p_{ji}}}$ for $p_{ji} \in [1,m]$; here we use $x_{p_{ji}}$ to indicate the occurrence of a variable in literal i of clause C_j. Given ϕ, we construct a relation schema R, an empty relation R_c, and a set $\Sigma \cup \{\varphi\}$ of CFD^{c}s defined on R, such that ϕ is satisfiable iff $\Sigma \not\models \varphi$.

(1) We define schema $R(C,V_c,X,V_x,Z)$, where $\mathsf{dom}(C) = \{1,\ldots,n\}$, $\mathsf{dom}(V_c) = \{\langle b_1b_2b_3\rangle \mid b_1,b_2,b_3 \in \{0,1\}\}$, $\mathsf{dom}(X) = \{x_1,\ldots,x_m\}$, which is the set of variables in ϕ, and moreover, both $\mathsf{dom}(V_x)$ and $\mathsf{dom}(Z)$ are integer. Intuitively, for each R tuple t, $t[C]$, $t[V_c]$, $t[X]$, $t[V_x]$ and $t[Z]$ specify a clause C, a truth assignment ξ (one of the eight to its three variables), one of the three variables in C, the truth value of the variable and the truth value of C determined by ξ.

(2) Let the set Σ of CFD^{c}s be $\Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \cup \Sigma_4$.

(a) Σ_1 encodes the relationships among attributes C, V_c, X and V_x. For each variable in a clause C_j ($1 \le j \le n$) and each value $\langle b_1b_2b_3\rangle$ in $\mathsf{dom}(V_c)$, there is a CFD^c in Σ_1. Thus there are $3*8$ CFD^{c}s for each clause C_j in Σ_1, and in total, there are $24*n$ CFD^{c}s

in Σ_1.

Each CFD^c for clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ is of the form of $R((C, V_c, X \to V_x), t_p, 1)$ such that $t_p[C] = j$, $t_p[V_c] = \langle b_1 b_2 b_3 \rangle$, and $t_p[X] = x_{p_{ji}}$ $(1 \leq i \leq 3)$. The value of $t_p[V_x]$ is decided by the value of $t_p[V_c]$ such that $t_p[V_x] = b_i$ if $y_{j_i} = x_{p_{ji}}$ and otherwise $t_p[V_x] = 1 - b_i$ if $y_{j_i} = \overline{x_{p_{ji}}}$.

For example, if $C_j = x_{p_{j1}} \vee \overline{x_{p_{j2}}} \vee x_{p_{j3}}$ such that $1 \leq p_{j1}, p_{j2}, p_{j3} \leq m$, then some possible pattern tuples are $(j, \langle 010 \rangle, x_{p_{j1}}, 0)$, $(j, \langle 010 \rangle, x_{p_{j2}}, 0)$, and $(j, \langle 010 \rangle, x_{p_{j3}}, 0)$.

(b) Σ_2 prevents certain variables from appearing in clauses. For each clause C_j and each variable x_i *not* in C_j, two CFD^{c}s are included in Σ_2: $\mu_{j,i,1} = R((C, X \to Z), (j, x_i \parallel 1), 1)$ and $\mu_{j,i,2} = R((C, X \to Z), (j, x_i \parallel 0), 1)$. Thus no tuple t satisfies $t[C] = j$ and $t[X] = x_i$, since otherwise $\mu_{j,i,1}$ forces $t[Z] = 1$ and $\mu_{j,i,2}$ forces $t[Z] = 0$. There are $(m-3) * n$ CFD^{c}s in Σ_2.

(c) Σ_3 encodes the relationship between the truth assignment V_c of clause C and its corresponding truth value Z of C. For clause C_j and each $h \in \mathsf{dom}(V_c)$, $\omega_h = R(V_c \to Z, t_{p_h}, 1)$ is in Σ_3, where $t_{p_h}[V_c] = h$, $t_{p_h}[Z] = 0$ if $h = \langle 000 \rangle$, *i.e., C* is not satisfied by the corresponding truth assignment h, and $t_{p_h}[Z] = 1$ otherwise. In total, Σ_3 consists of eight CFD^{c}s.

(d) Σ_4 includes $\mu_1 = R(C \to V_c, (\_ \parallel \_), 1)$ and $\mu_2 = R(X \to V_x, (\_ \parallel \_), 1)$, ensuring that for each clause C and each variable X, there is at most one truth assignment.
(3) CFD^c φ is defined as $R((Z \to C, X), (1 \parallel \_, \_), 3 * n - 1)$. Intuitively, φ assures that no more than $3 * n - 1$ tuples in an instance of R can have truth value 1 for their clauses.

Observe that Σ consists of $(m + 21) * n + 10$ CFD^{c}s. Thus the reduction is in PTIME.

We now show that ϕ is satisfiable iff $\Sigma \not\models \varphi$. Suppose first that ϕ is satisfiable. Then there exists a truth assignment ρ that makes ϕ true. Based on ρ, we construct an instance D of R with $3 * n$ tuples as follows. For each clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ and each variable $x_{p_{ji}}$ $(i \in [1,3])$ in C_j, we create a tuple t, where (a) $t[C] = j$; (b) $t[X] = x_{p_{ji}}$; (c) $t[Z] = 1$; (d) $t[V_x] = 1$ if $x_{p_{ji}}$ is assigned true by ρ, and otherwise $t[V_x] = 0$; (e) $t[V_c] = \langle b_1 b_2 b_2 \rangle$ such that for each $i \in [1,3]$, $b_i = 1$ if y_{j_i} is assigned true by ρ, and otherwise $b_i = 0$. That is, $t[V_c]$ is determined by ρ to all of its three variables. Observe that $D \models \Sigma$ but $D \not\models \varphi$. Hence $\Sigma \not\models \varphi$.

Conversely, if $\Sigma \not\models \varphi$, then there exists an instance D of R consisting of $3 * n$ tuples such that $D \models \Sigma$ but $D \not\models \varphi$. Observe that there exist at most n distinct values for attribute C, and each value of C can be associated with at most three distinct values of attribute X. Based on this, we define a truth assignment ρ such that

$\rho(x_i) = \mathsf{true}$ if $\pi_{V_x}(\sigma_{X=x_i}D) = \{1\}$ and $\rho(x_i) = \mathsf{false}$ otherwise. Observe that by $D \models \Sigma$, (a) $\pi_{V_x}(\sigma_{X=x_i}D)$ ($i \in [1,m]$) contains exactly one element, (b) $\pi_{V_c}(\sigma_{C=j}D)$ ($j \in [1,n]$) contains one element, and (c) $\pi_{CV_cV_x}(D)$ has $3*n$ elements. Indeed, since $D \models \Sigma$, the truth assignment ρ makes ϕ true. Thus ϕ is satisfiable. □

The proof of Theorem 3.3.5 actually yields a stronger result. Recall that a CFD^c $R(X \to Y, t_p, c)$ is a CFD of [FGJK08] when $c = 1$.

Corollary 3.3.6: *It remains co*NP*-complete to decide, given a set* Σ *of* CFD*s and a* CFD^c φ*, whether* $\Sigma \models \varphi$ *when* $\Sigma \cup \{\varphi\}$ *is bounded by a constant* $k = 3$*.* □

Not all is lost. Below we identify two tractable special cases. It should be remarked that while the second case below can find a counterpart for CFDs (Corollary 4.4 of [FGJK08]), its proof is quite different from that of [FGJK08]. Putting this and Corollary 3.3.6 together, one can tell that the extension of the equality operator and the presence of cardinality constraints take their toll in the implication analysis.

Proposition 3.3.7: *It is in* PTIME *to decide, given a set* $\Sigma \cup \{\varphi\}$ *of* CFD^c*s, whether* $\Sigma \models \varphi$ *when* $\Sigma \cup \{\varphi\}$ *is bounded by a constant* k *and one of the following conditions holds:*

(1) φ *is a* CFD *while* Σ *is a set of* CFD^c*s; or*

(2) Σ *is a set of* CFD*s,* φ *is a* CFD^c *and* $k = 0$*,* i.e., *all attributes in* Σ *or* φ *have an infinite domain.* □

Proof: Observe that $\Sigma \not\models \varphi$ iff there exists a nonempty instance D of the schema R on which Σ and φ are defined, such that $D \models \Sigma \cup \{\neg\varphi\}$. Thus it suffices to develop a PTIME algorithm to check the satisfiability of $\Sigma \cup \{\neg\varphi\}$.

Assume that φ is $R(X \to Y, t_p, c)$.

(1) Since φ is a CFD, the proof of Theorem 3.3.4 tells us that $\Sigma \cup \{\neg\varphi\}$ is satisfiable iff there exists an instance D_1 of R such that D_1 consists of two tuples t_1 and t_2, $D_1 \models \Sigma$, $t_1[X] \asymp t_p[X]$ and $t_1[X] \doteq t_2[X]$, but either $t_1[Y] \not\doteq t_2[Y]$, or there exists $l \in [1,2]$ such that $t_l[Y] \not\asymp t_p[Y]$. In light of these, a minor extension of the PTIME algorithm given in the proof of Proposition 3.3.2 suffices to check whether $\Sigma \cup \{\neg\varphi\}$ is satisfiable. Assume *w.l.o.g.* that Σ is defined on a schema R, and only attributes A_i in CFD^{c}s of Σ have a finite domain, for $i \in [1,k]$.

The algorithm checks whether there exists an instance $D_1 = \{t_1, t_2\}$ such that $D_1 \models \Sigma$, but $D_1 \not\models \varphi$. Initially, for each attribute $A \in X$, $t_1[A]$ and $t_2[A]$ are the same distinct variable x_A if $t_p[A]$ is '_', and $t_1[A] = t_2[A] = t_p[A]$ if $t_p[A]$ is a constant. For each other attribute A in $\mathtt{attr}(R)$ (but not in X), $t_1[A]$ and $t_2[A]$ are two distinct variables x_A and y_A,

respectively.

For all $i \in [1,k]$ and for each instantiation of variables x_{A_i} and y_{A_i} with values in $\mathsf{dom}(A_i)$, the algorithm does the following.

(a) For each CFD^c $\varphi' = R(X' \to Y', t'_p, c')$ in Σ, chase D_1 using φ'. If $t_i[X'] \asymp t'_p[X']$ ($i \in [1,2]$), then change $t_i[Y']$ such that $t_i[Y'] \asymp t'_p[Y']$, as long as there exists no attribute $A \in Y'$ such that $t_i[A]$ is already a constant that does not match $t'_p[A]$. Moreover, if $t_1[X'] \doteq t_2[X']$ and $c' \leq c$, then change $t_1[Y'] \doteq t_2[Y']$ as long as there exists no attribute $A \in Y'$ such that $t_1[A]$ and $t_2[A]$ are already constants and $t_1[A] \neq t_2[A]$. Here $c = 1$ since φ is a CFD.

(b) For each attribute $B \in \mathsf{attr}(R)$, if $t_i[B]$ ($i \in [1,2]$) is a variable after step (a), assign a distinct value from $\mathsf{dom}(B)$ to $t_i[B]$; note that $\mathsf{dom}(B)$ must be infinite in this case.

(c) If $D_1 \models \Sigma$ and $D_1 \not\models \varphi$, then return "yes".

The algorithm returns "no" if for all possible valuations to x_{A_i} and y_{A_i} for $i \in [1,k]$, it cannot instantiate D_1 such that $D_1 \models \Sigma$ but $D_1 \not\models \varphi$.

From these it follows that the algorithm returns "yes" iff $\Sigma \not\models \varphi$. In addition, similar to the proof of Proposition 3.3.2, it is easy to see that the algorithm is in PTIME in the sizes of Σ, φ, relation R_c (in the definition of $\doteq$), and the maximum cardinality of the k finite domains.

(2) A PTIME algorithm similar to the one given in the proof of (1) suffices to check whether $\Sigma \cup \{\neg\varphi\}$ is satisfiable. Here the algorithm operates on $c+1$ tuples, as described in the proof of Theorem 3.3.4. Since Σ consists of CFDs only, the chase of the tuples using CFDs in Σ is straightforward. Since all the attributes in Σ or φ have an infinite domain, we no longer need to check valuations to those variables denoting attributes with a finite domain. One can verify that the algorithm is in PTIME. □

3.4 Extending CFDs and CINDs with Built-in Predicates

In this section, we propose a natural extension of conditional functional dependencies (CFDs) and conditional inclusion dependencies (CINDs), denoted by CFD^{p}s and CIND^{p}s, respectively, by specifying patterns of data values with $\neq, <, \leq, >$ and $\geq$ predicates. And we study the satisfiability problems and the implication problems of both CFD^{p}s and CIND^{p}s.

	id	name	type	price	shipping	sale	state
t_1:	b1	Harry Potter	book	25.99	0	T	WA
t_2:	c1	Snow White	CD	9.99	2	F	NY
t_3:	b2	Catch-22	book	34.99	20	F	DL
t_4:	a1	Sunflowers	art	5m	500	F	DL

(a) An item relation

	state	rate
t_5:	PA	6
t_6:	NY	4
t_7:	DL	0
t_8:	NJ	3.5

(b) tax rates

Figure 3.4: Example instance D_0 of item and tax

3.4.1 Motivation Example

Extensions of functional dependencies (FDs) and inclusion dependencies (INDs), known as *conditional functional dependencies* (CFDs [FGJK08]) and *conditional inclusion dependencies* (CINDs [BFM07]), respectively, have recently been proposed for improving data quality. These extensions enforce patterns of semantically related data values, and detect errors as violations of the dependencies. Conditional dependencies are able to capture more inconsistencies than FDs and INDs [FGJK08, BFM07].

Conditional dependencies specify constant patterns in terms of equality $(=)$. In practice, however, the semantics of data often needs to be specified in terms of other predicates such as $\neq, <, \leq, >$ and $\geq$, as illustrated by the example below.

Example 3.4.1: An online store maintains a database of two relations: (a) item for items sold by the store, and (b) tax for the sale tax rates for the items, except artwork, in various states. The relations are specified by the following schemas:

item (id: string, name: string, type: string, price: float, shipping: float,
sale: bool, state: string)
tax (state: string, rate: float)

where each item is specified by its id, name, type (*e.g.,* book, CD), price, shipping fee, the state to which it is shipped, and whether it is on sale. A tax tuple specifies the sale tax rate in a state. An instance D_0 of item and tax is shown in Fig. 3.4.

One wants to specify dependencies on the relations as data quality rules to detect errors in the data, such that inconsistencies emerge as violations of the dependencies. Traditional dependencies (FDs, INDs; see, *e.g.,* [AHV95]) and conditional dependencies (CFDs, CINDs [FGJK08, BFM07]) on the data include the following:

cfd_1: item (id $\rightarrow$ name, type, price, shipping, sale)
cfd_2: tax (state $\rightarrow$ rate)
cfd_3: item (sale = 'T' $\rightarrow$ shipping = 0)

These are CFDs: (a) cfd_1 assures that the id of an item uniquely determines the name, type, price, shipping, sale of the item; (b) cfd_2 states that state is a key for tax, *i.e.,* for each state there is a unique sale tax rate; and (c) cfd_3 is to ensure that for any item tuple t, if $t[\mathsf{sale}] =$ 'T' then $t[\mathsf{shipping}]$ must be 0; *i.e.,* the store provides free shipping for items on sale. Here cfd_3 is specified in terms of patterns of semantically related data values, namely, sale = 'T' and shipping = 0. It is to hold only on item tuples that match the pattern sale = 'T'. In contrast, cfd_1 and cfd_2 are traditional FDs without constant patterns, a special case of CFDs. One can verify that no sensible INDs or CINDs can be defined across item and tax.

Note that D_0 of Fig. 3.4 satisfies cfd_1, cfd_2 and cfd_3. That is, when these dependencies are used as data quality rules, no errors are found in D_0.

In practice, the shipment fee of an item is typically determined by the price of the item. Moreover, when an item is on sale, the price of the item is often in a certain range. Furthermore, for any item sold by the store to a customer in a state, if the item is *not* artwork, then one expects to find the sale tax rate in the state from the tax table. These semantic relations cannot be expressed as CFDs of [FGJK08] or CINDs of [BFM07], but can be expressed as the following dependencies:

pfd_1: item (sale $=$ 'F' **and** price $\leq 20 \rightarrow$ shipping = 3)
pfd_2: item (sale $=$ 'F' **and** price > 20 **and** price $\leq 40 \rightarrow$ shipping = 6)
pfd_3: item (sale $=$ 'F' **and** price $> 40 \rightarrow$ shipping = 10)
pfd_4: item (sale $=$ 'T' $\rightarrow$ price ≥ 2.99 **and** price < 9.99)
pind_1: item (state; type $\neq$ 'art') $\subseteq$ tax (state; nil)

Here pfd_2 states that for any item tuple, if it is not on sale and its price is in the range $(20, 40]$, then its shipment fee must be 6; similarly for pfd_1 and pfd_3. These dependencies extend CFDs [FGJK08] by specifying patterns of semantically related data values in terms of predicates $<, \leq, >$, and $\geq$. Similarly, pfd_4 assures that for any item tuple, if it is on sale, then its price must be in the range $[2.99, 9.99)$. Dependency pind_1 extends CINDs [BFM07] by specifying patterns with $\neq$: for any item tuple t, if $t[\mathsf{type}]$ is *not* artwork, then there must exist a tax tuple t' such that $t[\mathsf{state}] = t'[\mathsf{state}]$, *i.e.,* the sale tax of the item can be found from the tax relation.

Using pfd_1–pfd_4 and pind_1 as data quality rules, we find that D_0 of Fig. 3.4 is *not* clean. Indeed, (a) t_2 violates pfd_1: its price is less than 20, but its shipping fee is 2 rather than 3; similarly, t_3 violates pfd_2, and t_4 violates pfd_3. (b) Tuple t_1 violates pfd_4: it is on sale but its price is not in the range $[2.99, 9.99)$. (c) The database D_0 also

violates pind_1: t_1 is not artwork, but its state cannot find a match in the tax relation, *i.e.,* no tax rate for WA is found in D_0. □

None of pfd_1–pfd_4 and pind_1 can be expressed as FDs or INDs [AHV95], which do not allows constants, or as CFDs [FGJK08] or CINDs [BFM07], which specify patterns with equality ($=$) only. While there have been extensions of CFDs [BFGM08, GKK+08], none of these allows dependencies to be specified with patterns on data values in terms of built-in predicates $\neq, <, \leq, >$ or $\geq$. To the best of our knowledge, no previous work has studied extensions of CINDs.

These highlight the need for extending CFDs and CINDs to capture errors commonly found in real-life data. While one can consider arbitrary extensions, it is necessary to strike a balance between the expressive power of the extensions and their complexity. In particular, we want to be able to reason about data quality rules expressed as extended CFDs and CINDs. Furthermore, we want to have effective algorithms to detect inconsistencies based on these extensions.

3.4.2 Incorporating Built-in Predicates into CFDs

We now define CFD^{p}s, also referred to as *conditional functional dependencies*, by extending CFDs with predicates ($\neq, <, \leq, >, \geq$) in addition to equality ($=$).

Consider a relation schema R defined over a finite set of attributes, denoted by $\mathsf{attr}(R)$. For each attribute $A \in \mathsf{attr}(R)$, its domain is specified in R, denoted as $\mathsf{dom}(A)$, which is either finite (*e.g.,* bool) or infinite (*e.g.,* string). We assume *w.l.o.g.* that a domain is totally ordered if $<, \leq, >$ or $\geq$ is defined on it.

Syntax. A CFD^p φ on R is a pair $R(X \to Y,\ T_p)$, where (1) X, Y are sets of attributes in $\mathsf{attr}(R)$; (2) $X \to Y$ is a standard FD, referred to as the FD *embedded in* φ; and (3) T_p is a tableau with attributes in X and Y, referred to as the *pattern tableau* of φ, where for each A in $X \cup Y$ and each tuple $t_p \in T_p$, $t_p[A]$ is either an unnamed variable '_' that draws values from $\mathsf{dom}(A)$, or '$\mathsf{op}\ a$', where op is one of $=, \neq, <, \leq, >, \geq$, and '$a$' is a constant in $\mathsf{dom}(A)$.

If attribute A occurs in both X and Y, we use A_L and A_R to indicate the occurrence of A in X and Y, respectively, and separate the X and Y attributes in a pattern tuple with '||'. We write φ as $(X \to Y,\ T_p)$ when R is clear from the context, and denote X as $\mathsf{LHS}(\varphi)$ and Y as $\mathsf{RHS}(\varphi)$.

Example 3.4.2: The dependencies cfd1–cfd3 and pfd1–pfd4 that we have seen in Example 3.4.1 can all be expressed as CFD^{p}s. Figure 3.5 shows some of these CFD^{p}s: φ_1 (for FD cfd_2), φ_2 (for CFD cfd_3), φ_3 (for pfd_2), and φ_4 (for pfd_4). □

(1) φ_1 = tax (state → rate, T_1)

	state	rate
T_1:	_	_

(2) φ_2 = item (sale → shipping, T_2)

	sale	shipping
T_2:	= T	= 0

(3) φ_3 = item (sale, price →shipping, T_3)

	sale	price	shipping
	= F	> 20	= 6
T_3:	= F	≤ 40	= 6

(4) φ_4 = item (sale → price, T_4)

	sale	price
	= T	≥ 2.99
T_4:	= T	< 9.99

Figure 3.5: Example CFD^{p}s

Semantics. Consider CFD^p $\varphi = (R : X \to Y,\ T_p)$, where $T_p = \{t_{p1}, \ldots, t_{pk}\}$.

A data tuple t of R is said to *match* LHS(φ), denoted by $t[X] \asymp T_p[X]$, if *for each tuple* t_{pi} *in* T_p and *each attribute* A in X, either (a) $t_{pi}[A]$ is the wildcard '_' (which matches any value in dom(A)), or (b) $t[A]$ op a if $t_{pi}[A]$ is 'op a', where the operator op ($=, \neq, <, \leq, >$ or $\geq$) is interpreted by its standard semantics. Similarly, the notion that t *matches* RHS(φ) is defined, denoted by $t[Y] \asymp T_p[Y]$.

Intuitively, each pattern tuple t_{pi} specifies a condition via $t_{pi}[X]$, and $t[X] \asymp T_p[X]$ if $t[X]$ satisfies the *conjunction* of all these conditions. Similarly, $t[Y] \asymp T_p[Y]$ if $t[Y]$ matches all the patterns specified by $t_{pi}[Y]$ for all t_{pi} in T_p.

An instance I of R *satisfies* the CFD^p φ, denoted by $I \models \varphi$, if for *each pair* of tuples t_1, t_2 in the instance I, if $t_1[X] = t_2[X] \asymp T_p[X]$, then $t_1[Y] = t_2[Y] \asymp T_p[Y]$. That is, if $t_1[X]$ and $t_2[X]$ are equal and in addition, they both match the pattern tableau $T_p[X]$, then $t_1[Y]$ and $t_2[Y]$ must also be equal to each other and they both match the pattern tableau $T_p[Y]$.

Observe that φ is imposed only on the subset of tuples in I that match LHS(φ), rather than on the entire I. For all tuples t_1, t_2 in this subset, if $t_1[X] = t_2[X]$, then (a) $t_1[Y] = t_2[Y]$, *i.e.,* the semantics of the embedded FDs is enforced; and (b) $t_1[Y] \asymp T_p[Y]$, which assures that the *constants* in $t_1[Y]$ match the *constants* in $t_{pi}[Y]$ for all t_{pi} in T_p. Note that here tuples t_1 and t_2 can be the same.

An instance I of R satisfies a set Σ of CFD^{p}s, denoted by $I \models \Sigma$, if $I \models \varphi$ for *each* CFD^p φ in Σ.

Example 3.4.3: The instance D_0 of Fig. 3.4 satisfies φ_1 and φ_2 of Fig. 3.5, but neither φ_3 nor φ_4. Indeed, tuple t_3 violates (*i.e.,* does not satisfy) φ_3, since t_3[sale] = 'F' *and* $20 < t_3$[price] ≤ 40, but t_3[shipping] is 20 instead of 6. Note that t_3 matches LHS(φ_3) since it satisfies the condition specified by the *conjunction* of the pattern tuples in T_3. Similarly, t_1 violates φ_4, since t_1[sale] = 'T' but t_1[price] > 9.99. Observe that while it takes two tuples to violate a standard FD, a single tuple may violate a CFD^p. □

Special cases. (1) A standard FD $X \to Y$ [AHV95] can be expressed as a CFD ($X \to$

(1) ψ_1 = (item [state; type] $\subseteq$ tax [state; *nil*], T_1),

(2) ψ_2 = (item [state; type, state] $\subseteq$ tax [state; rate], T_2)

T_1:

type	nil
$\neq$ art	

T_2:

type	state	rate
$\neq$ art	$=$ DL	$=0$

Figure 3.6: Example CIND^{p}s

Y, T_p) in which T_p contains a single tuple consisting of '_' only, without constants. (2) A CFD ($X \to Y$, T_p) [FGJK08] with $T_p = \{t_{p1}, \ldots, t_{pk}\}$ can be expressed as a set $\{\varphi_1, \ldots, \varphi_k\}$ of CFD^{p}s such that for $i \in [1,k]$, $\varphi_i = (X \to Y, T_{pi})$, where T_{pi} contains a single pattern tuple t_{pi} of T_p, with equality ($=$) only. For example, φ_1 and φ_2 in Fig. 3.5 are CFD^{p}s representing FD cfd2 and CFD cfd3 in Example 3.4.1, respectively. Note that all data quality rules in [CM08, GKK$^+$08] can be expressed as CFD^{p}s.

3.4.3 Incorporating Built-in Predicates into CINDs

Along the same lines as CFD^{p}s, we next define CIND^{p}s, also referred to as *conditional inclusion dependencies*. Consider two relation schemas R_1 and R_2.

Syntax. A CIND^p ψ is a pair $(R_1[X; X_p] \subseteq R_2[Y; Y_p], T_p)$, where (1) X, X_p and Y, Y_p are lists of attributes in $\mathsf{attr}(R_1)$ and $\mathsf{attr}(R_2)$, respectively; (2) $R_1[X] \subseteq R_2[Y]$ is a standard IND, referred to as the IND *embedded* in ψ; and (3) T_p is a tableau, called the *pattern tableau* of ψ defined over attributes $X_p \cup Y_p$, and for each A in X_p or Y_p and each pattern tuple $t_p \in T_p$, $t_p[A]$ is either an unnamed variable '_' that draws values from $\mathsf{dom}(A)$, or 'op a', where op is one of $=, \neq, <, \leq, >, \geq$ and 'a' is a constant in $\mathsf{dom}(A)$.

We denote $X \cup X_p$ as $\mathsf{LHS}(\psi)$ and $Y \cup Y_p$ as $\mathsf{RHS}(\psi)$, and separate the X_p and Y_p attributes in a pattern tuple with '$\|$'. We use nil to denote an *empty* list.

Example 3.4.4: Figure 3.6 shows two example CIND^{p}s: ψ_1 expresses pind_1 of Example 3.4.1, and ψ_2 refines ψ_1 by stating that for any item tuple t_1, if its type is not art and its state is DL, then there must be a tax tuple t_2 such that its state is DL and rate is 0, *i.e.,* ψ_2 assures that the sale tax rate in Delaware is 0. □

Semantics. Consider CIND^p $\psi = (R_1[X; X_p] \subseteq R_2[Y; Y_p], T_p)$. An instance (I_1, I_2) of (R_1, R_2) *satisfies* the CIND^p ψ, denoted by $(I_1, I_2) \models \psi$, iff for *each* tuple $t_1 \in I_1$, if $t_1[X_p] \asymp T_p[X_p]$, then there *exists* a tuple $t_2 \in I_2$ such that $t_1[X] = t_2[Y]$ and moreover, $t_2[Y_p] \asymp T_p[Y_p]$.

That is, if $t_1[X_p]$ matches the pattern tableau $T_p[X_p]$, then ψ requires the existence of t_2 such that (1) $t_1[X] = t_2[Y]$ as required by the standard IND embedded in ψ; and (2) $t_2[Y_p]$ must match the pattern tableau $T_p[Y_p]$. In other words, ψ is "conditional" since its embedded IND is applied only to the subset of tuples in I_1 that match $T_p[X_p]$,

and moreover, the pattern $T_p[Y_p]$ is enforced on the tuples in I_2 that match those tuples in I_1. As remarked in Section 3.4.2, the pattern tableau T_p specifies the *conjunction* of patterns of all tuples in T_p.

Example 3.4.5: The instance D_0 of item and tax in Fig. 3.4 violates CIND^p ψ_1. Indeed, tuple t_1 in item *matches* $\mathsf{LHS}(\psi_1)$ since $t_1[\mathsf{type}] \neq$ 'art', but there is no tuple t in tax such that $t[\mathsf{state}] = t_1[\mathsf{state}] =$ 'WA'. In contrast, D_0 satisfies ψ_2. □

We say that a database D satisfies a set Σ of CINDs, denoted by $D \models \Sigma$, if $D \models \varphi$ for each $\varphi \in \Sigma$.

Safe CIND^{p}s. Consider CIND^p $\psi = (R_1[X;\ X_p] \subseteq R_2[Y;\ Y_p],\ T_p)$. We say a CIND^p $(R_1[X;\ X_p] \subseteq R_2[Y;\ Y_p],\ T_p)$ is *unsafe* if there exists a pattern tuple t_p in T_p such that (a) there exist $A \in X$ and $B \in Y$ with A corresponding to B in the IND and (b) $t_p[A] \neq t_p[B] \wedge t_p[A]$ (recall that here $t_p[A]$ and $t_p[B]$ are conditions). Consider the case when both attributes A and B are price, and $t_p[\mathsf{A}] = 9.99$ and $t_p[\mathsf{B}] \geq 19.99$. There exists no nonempty database that satisfies such unsafe CIND^{p}s. Indeed the constraints $t_p[A]$ and $t_p[B]$ are on the same data value, and, thus, unsafe CIND^{p}s do not make sense. It takes $O(|T_p|^2)$-time in the size $|T_p|$ of T_p to decide whether a CIND^p is unsafe. Thus in the sequel we consider safe CIND^p only.

Special cases. Observe that (1) a standard CIND $(R_1[X] \subseteq R_2[Y])$ can be expressed as a CIND^p $(R_1[X;\ nil] \subseteq R_2[Y;\ nil],\ T_p)$ such that T_p is simply an *empty* set; and (2) a CIND $(R_1[X;\ X_p] \subseteq R_2[Y;\ Y_p],\ T_p)$ with $T_p = \{t_{p1}, \ldots, t_{pk}\}$ can be expressed as a set $\{\psi_1, \ldots, \psi_k\}$ of CIND^{p}s, where for $i \in [1,k]$, $\psi_i = (R_1[X;\ X_p] \subseteq R_2[Y;\ Y_p],\ T_{pi})$ such that T_{pi} consists of a single pattern tuple t_{pi} of T_p defined in terms of equality ($=$) only.

3.4.4 Reasoning about CFD^{p}s and CIND^{p}s

The satisfiability problem and the implication problem are the two central technical questions associated with any dependency languages. In this section we investigate these problems for CFD^{p}s and CIND^{p}s, separately and taken together.

3.4.5 The Satisfiability Analysis

The satisfiability problem is to determine, given a set Σ of constraints, whether there exists a *nonempty* database that satisfies Σ.

The satisfiability analysis of conditional dependencies is not only of theoretical interest, but is also important in practice. Indeed, when CFD^{p}s and CIND^{p}s are used as

data quality rules, this analysis helps one check whether the rules make sense themselves. The need for this is particularly evident when the rules are manually designed or discovered from various datasets [CM08, GKK+08, FGLX].

The satisfiability analysis of CFD^{p}s. Given any FDs, one does not need to worry about their satisfiability since any set of FDs is always satisfiable. However, as observed in [FGJK08], for a set Σ of CFDs on a relational schema R, there may not exist a *nonempty* instance I of R such that $I \models \Sigma$. As CFDs are a special case of CFD^{p}s, the same problem exists when it comes to CFD^{p}s.

Example 3.4.6: Consider CFD^p $\varphi = (R : A \to B,\ T_p)$ such that $T_p = \{(\_ \,\|\, = a), (\_ \,\|\, \neq a)\}$. Then there exists no *nonempty* instance I of R that satisfies φ. Indeed, for any tuple t of R, φ requires that both $t[B] = a$ and $t[B] \neq a$. □

This problem is already NP-complete for CFDs [FGJK08]. Below we show that it has the same complexity for CFD^{p}s despite their increased expressive power.

Theorem 3.4.1: *The satisfiability problem for* CFD^p*s is* NP*-complete.* □

Proof: The lower bound follows from the NP-hardness of their CFDs counterparts [FGJK08], since CFDs are a special case of CFD^{p}s.

We next show that the problem is in NP by presenting an NP algorithm that, given a set Σ of CFD^{p}s defined on a relational schema R, determines whether Σ is satisfiable. The satisfiability problem has the following small model property: if there exists a nonempty instance I of R such that $I \models \Sigma$, then for any tuple $t \in I$, $I_t = \{t\}$ is an instance of R and $I_t \models \Sigma$. Thus it suffices to consider single-tuple instances for deciding whether Σ is satisfiable.

Assume *w.l.o.g.* that the attributes $\mathsf{attr}(R) = \{A_1, \ldots, A_m\}$ and the total number of pattern tuples of all pattern tableaux T_p in Σ is h. For each $i \in [1, m]$, define the active domain of attribute A_i to be a set $\mathsf{adom}(A_i) = C_i^0 \cup C_i^1$, where (a) the set C_i^0 consists of all constants in $T_p[A_i]$ of all pattern tableaux T_p in Σ and let $C_i^0 = \{a_1\}$, where $a_1 \in \mathsf{dom}(A_i)$, if C_i^0 is empty, and (b) the set C_i^1 contains an extra distinct constant not appearing in C_i^0 for those attributes whose domains are not totally ordered, *i.e.,* involving no predicates $<, \leq, >$ and $\geq$; otherwise, it is constructed for those attributes whose domains have total orders as below.

- Arrange all constants in C_i^0 in the increasing order, and assume that the resulting $C_i^0 = \{a_1, \ldots, a_k\}$, where $k \geq 1$.
- Add a constant $b_{01} \in \mathsf{dom}(A_i)$ to C_i^1 such that $b_{01} < a_1$ if there exists one; and add another constant $b_{02} \in \mathsf{dom}(A_i)$ to C_i^1 such that $b_{02} < a_1$ and $b_{02} \neq b_{01}$ if

there exists one.

- Similarly, for each $j \in [1, k-1]$, add a constant $b_{j1} \in \mathsf{dom}(A_i)$ to C_i^1 such that $a_j < b_{j1} < a_{j+1}$ if there exists one; and add another constant $b_{j2} \in \mathsf{dom}(A_i)$ to C_i^1 such that $a_j < b_{j2} < a_{j+1}$ and $b_{j2} \neq b_{j1}$ if there exists one.
- Add a constant $b_{k1} \in \mathsf{dom}(A_i)$ to C_i^1 such that $b_{k1} > a_k$ if there exists one; and add another constant $b_{k2} \in \mathsf{dom}(A_i)$ to C_i^1 such that $b_{k2} > a_k$ and $b_{k2} \neq b_{k1}$ if there exists one.

Observe that the number of elements in $\mathsf{adom}(A_i)$ is at most $O(3 * h + 2)$.

One can easily verify that $I = \{t\} \models \Sigma$ if and only if $I_\rho \models \Sigma$, where $I_\rho = \{(\rho(t[A_1]), \ldots, \rho(t[A_m]))\}$. Here ρ is a mapping from $t[A_i]$ to $\mathsf{adom}(A_i) = C_i^0 \cup C_i^1$ for each $i \in [1, m]$ such that (a) $\rho(t[A_i]) = \rho(t[A_i])$ when $t[A_i] \in C_i^0$, (b) $\rho(t[A_i]) = b_{01}$ when $t[A_i] < a_1$, (c) $\rho(t[A_i]) = b_{k1}$ when $t[A_i] > a_k$, and (d) $\rho(t[A_i]) = b_{j1}$ when $a_j < t[A_i] < a_{j+1}$.

Based on these, we give an NP algorithm as follows:

(1) Guess a single tuple t of R such that $t[A_i] \in \mathsf{adom}A_i$ for each $i \in [1, m]$.

(2) Check whether $I = \{t\} \models \Sigma$, and return 'yes' if so.

Obviously both steps (1) and (2) can be done in PTIME in the size of Σ, and it is easy to verify that $\Sigma \models \varphi$ if and only if the algorithm returns 'yes'. Hence the algorithm is in NP, and so is the problem. □

It is known [FGJK08] that the satisfiability problem for CFDs is in PTIME when the CFDs considered are defined over attributes that have an infinite domain, *i.e.,* in the absence of finite domain attributes. However, this is no longer the case for CFD^{p}s. This tells us that the increased expressive power of CFD^{p}s does take a toll in this special case. It should be remarked that while the proof of Proposition 3.4.1 is an extension of its counterpart in [FGJK08], the result below is new.

Theorem 3.4.2: *In the absence of finite domain attributes, the satisfiability problem for* CFD^p*s remains* NP*-complete.* □

Proof: The problem is in NP following from Theorem 3.4.1.

We next show that the problem is NP-hard by reduction from the 3SAT problem, which is NP-complete (cf. [GJ79]). Consider an instance $\phi = C_1 \wedge \cdots \wedge C_n$ of 3SAT, where all the variables in ϕ are $x_1, \ldots, x_m$, C_j is of the form $y_{j_1} \vee y_{j_2} \vee y_{j_3}$, and moreover, for $i \in [1, 3]$, y_{j_i} is either $x_{p_{ji}}$ or $\overline{x_{p_{ji}}}$ for $p_{ji} \in [1, m]$. Given the instance ϕ, we construct a relational schema R and a set Σ of CFD^{p}s defined on R such that ϕ is satisfiable if and only if the Σ is satisfiable.

(1) Define the relation $R(X_1,\ldots,X_m,C_1,\ldots,C_n,Z)$, where all attributes share a totally ordered infinite domain dom, and there exists a constant $a \in \mathsf{dom}$.

Intuitively, for each R tuple t, $t[X_1, \ldots, X_m]$ specifies a truth assignment ξ for variables $x_1,\ldots,x_m$ of ϕ, and $t[C_i]$ and $t[Z]$ are the truth values of the clause C_i and the sentence ϕ *w.r.t.* the truth assignment ξ, respectively.

(2) Let the set Σ of CFD^{p}s be $\Sigma_0 \cup \Sigma_1 \cup \ldots \cup \Sigma_n \cup \Sigma_{n+1}$.

- The Σ_0 contains $n+1$ CFD^{p}s. Intuitively, the $\Sigma_0 = \{\varphi_0, \varphi_1, \ldots, \varphi_n\}$ encodes the relationships between the truth values of clauses $C_1,\ldots,C_n$ and the truth value of sentence ϕ.

  For each clause C_i ($i \in [1,n]$), we define the CFD^p $\varphi_i = (C_1,\ldots,C_n \to Z, T_{pi})$ such that $T_{pi} = \{t_{pi}\}$ and $t_{pi}[C_i,Z] = (\neq a \,\|\neq a)$ and $t_{pi}[C_j] =$ '_' ($j \neq i, j \in [1,n]$). Moreover, we define the CFD^p $\varphi_0 = (C_1,\ldots,C_n \to Z, T_{p0})$, where $T_{p0} = \{(= a,\ldots,= a \,\|= a)\}$. Intuitively, we use $\neq a$ and $= a$ to represent the truth values false and true for the clauses and the sentence.
- The Σ_i contains 8 CFD^{p}s for $i \in [1,n]$. Intuitively, the $\Sigma_i = \{\varphi_{i,0},\ldots,\varphi_{i,7}\}$ encodes the relationships between the truth values of the three variables in clause C_i.

  Consider clause $C_i = x_{j_1} \vee \overline{x_{j_2}} \vee x_{j_3}$ of ϕ, where $1 \leq j_1, j_2, j_3 \leq m$, we define CFD^{p}s $\varphi_{i,0} = (X_{j_1},X_{j_2},X_{j_3} \to C_i, T_{pi,0})$, where $T_{pi,0} = \{(< a, < a, < a \,\|= a)\}$, and $\varphi_{i,2} = (X_{j_1},X_{j_2},X_{j_3} \to C_i, T_{pi,2})$, where $T_{pi,2} = \{(< a, \geq a, < a \,\|\neq a)\}$. Intuitively, we use $< a$ and $\geq a$ to represent the truth values false and true for variables, respectively.

  Similarly, we can define the rest 6 CFD^{p}s $\varphi_{i,1}$, $\varphi_{i,3}$, $\varphi_{i,4}$, $\varphi_{i,5}$, $\varphi_{i,6}$ and $\varphi_{i,7}$.
- The Σ_{n+1} contains a single CFD^p $\varphi_{n+1} = (Z \to Z, T_{p(n+1)})$, where $T_{p(n+1)} = \{(\_ \,\|= a)\}$. Intuitively, φ_{n+1} enforces that for any R tuple t, $t[Z] = a$.

Observe that Σ contains $8 * m + n + 2$ CFD^{p}s. Thus the reduction is in PTIME.

We now show that the ϕ is satisfiable if and only if the Σ is satisfiable.

Suppose first that the Σ is satisfiable, then there exists a nonempty instance I of R such that $I \models \Sigma$. For any tuple $t \in I$, (a) the Σ_{n+1} forces $t[Z] = a$, (b) the Σ_0 forces $t[C_1,\ldots,C_n] = (a,\ldots,a)$, and (c) for each clause C_i ($i \in [1,n]$) with variables x_{j_1}, x_{j_2} and x_{j_3}, the Σ_j forces that $t[X_{j_1},X_{j_2},X_{j_3}]$ does not match the LHS of the CFD^p that forces $t[C_i] \neq a$. From the tuple t, we can construct a truth assignment ξ of ϕ such that $\xi(x_i)$ = false if $t[X_i] < a$ and $\xi(x_i)$ = true if $t[X_i] \geq a$ ($i \in [1,m]$). Since $\{t\} \models \Sigma$, it is easy to verify that the truth assignment ξ makes ϕ true.

Conversely, suppose that the ϕ is satisfiable. Then there exists a truth assignment ξ that makes ϕ true. We construct an R tuple t as follows: (a) $t[C_1,\ldots,C_n,Z] = (a,\ldots,a)$

and (b) for each $i \in [1,m]$, $t[X_i] = a_i$ such that $a_i \in \mathsf{dom}$, $a_i \geq a$ if $\xi(x_i) = \mathsf{true}$ and $a_i < a$ otherwise. Let $I = \{t\}$, then one can easily verify that $I \models \Sigma$. □

The satisfiability analysis of CIND^{p}s. Like FDs, one can specify arbitrary INDs or CINDs without worrying about their satisfiability. Below we show that CIND^{p}s also have this property, by extending the proof of its counterpart in [BFM07] and Chapter [BFM07].

Proposition 3.4.3: *Any set Σ of CIND^{p}s is always satisfiable.* □

Proof: It suffices to show that given a set Σ of CIND^{p}s over a database schema $\mathcal{R}(R_1,\ldots,R_n)$, we can construct a *nonempty* instance $D = (I_1,\ldots,I_n)$ of $\mathcal{R}$ such that $D \models \Sigma$. We build D as follows. First, for each attribute A, define the active domain of A to be a set $\mathsf{adom}(A)$, which consists of certain data values in $\mathsf{dom}(A)$. Second, we construct the D by using these active domains.

We start with the construction of active domains. (a) For each attribute A, first initialize $\mathsf{adom}(A)$ along the same lines as the one for CFD^{p}s in Theorem 3.4.1. Then (b) for each CIND^p $(R_a[A_1,A_2,\ldots,A_m;X_p] \subseteq R_b[B_1,\ldots,B_m;Y_p],T_p)$ in Σ, let $\mathsf{adom}(B_i) = \mathsf{adom}(B_i) \cup \mathsf{adom}(A_i)$ for each $i \in [1,m]$. This rule is repeatedly applied until a fixpoint of $\mathsf{adom}(A)$ is reached for all attributes A. It is easy to verify that this process terminates since we start with a finite set of data values.

We next construct the D. For each relation $R_i(A_1,\ldots,A_k) \in \mathcal{R}$, we define $I_i = \mathsf{adom}(A_1) \times \cdots \times \mathsf{adom}(A_k)$, where $\times$ is the *Cartesian product* operation [Ram98]. And we define $D = \{I_1,\ldots,I_n\}$.

It is easy to verify that D is nonempty and $D \models \Sigma$. □

The satisfiability analysis of CFD^{p}s and CIND^{p}s. The satisfiability problem for CFDs and CINDs taken together is undecidable [BFM07]. Since CFD^{p}s and CIND^{p}s subsume CFDs and CINDs, respectively, from these we immediately have:

Corollary 3.4.4: *The satisfiability problem for CFD^{p}s and CIND^{p}s is undecidable.* □

3.4.6 The Implication Analysis

The implication problem is to determine, given a set Σ of dependencies and another dependency ϕ, whether or not Σ entails ϕ, denoted by $\Sigma \models \phi$. That is, whether or not for all databases D, if $D \models \Sigma$ then $D \models \phi$.

The implication analysis helps us remove redundant data quality rules, and thus improve the performance of error detection and repairing based on the rules.

Example 3.4.7: The CFD^{p}s of Fig. 3.5 imply CFD^{p}s $\varphi = \mathsf{item}\,(\mathsf{sale},\mathsf{price} \rightarrow \mathsf{shipping},$

T), where T consists of a single pattern tuple (sale ='F', price $= 30 \parallel$ shipping $= 6$). Thus in the presence of the CFD^{p}s of Fig. 3.5, φ is redundant. □

The implication analysis of CFD^{p}s. We first show that the implication problem for CFD^{p}s retains the same complexity as their CFDs counterpart. The result below is verified by extending the proof of its counterpart in [FGJK08].

Theorem 3.4.5: *The implication problem for* CFD^p*s is co*NP*-complete.* □

Proof: The lower bound follows from the coNP-hardness of their CFDs counterparts [FGJK08], since CFDs are a special case of CFD^{p}s.

We show that the problem is in coNP by presenting an NP algorithm for its complement problem, *i.e.,* for determining whether $\Sigma \not\models \varphi$. The algorithm is based on a small model property of CFD^{p}s: if $\varphi = (R : X \to Y, T_p)$ and $\Sigma \not\models \varphi$, then there exists an instance I of R with two tuples t_1 and t_2 such that $I \models \Sigma$ and $t_1[X] = t_2[X] \asymp T_p[X]$, but either $t_1[Y] \neq t_2[Y]$ or $t_1[Y] \not\asymp T_p[Y]$ (resp. $t_2[Y] \not\asymp T_p[Y]$). Thus it suffices to consider instances I with two tuples for deciding whether $\Sigma \not\models \varphi$.

Assume *w.l.o.g.* that the attributes $\mathsf{attr}(R) = \{A_1, \ldots, A_m\}$. For each $i \in [1,m]$, define the active domain of attribute A_i to be the set $\mathsf{adom}(A_i) = C_i^0 \cup C_i^1$ defined in the proof Theorem 3.4.1. Then one can easily verify that $I = \{t_1, t_2\} \models \Sigma$ and $I = \{t_1, t_2\} \not\models \varphi$ if and only if $I' \models \Sigma$ and $I' \not\models \varphi$, where $I' = \{(\rho_1(t_1[A_1]), \ldots, \rho_1(t_2[A_m])), (\rho_2(t_2[A_1]), \ldots, \rho_2(t_2[A_m]))\}$. Here ρ_1 (resp. ρ_2) is a mapping from $t_1[A_i]$ (resp. $t_2[A_i]$) to $\mathsf{adom}(A_i)$ ($i \in [1,m]$), where

(a) $\rho_1(t_1[A_i]) = \rho_1(t_1[A_i])$ when $t_1[A_i] \in C_0$, and $\rho_2(t_2[A_i]) = \rho_2(t_2[A_i])$ when $t_2[A_i] \in C_0$;
(b) for each attribute A_i ($i \in [1,m]$), $\rho_1(t_1[A_i]) = b_{01}$ when $t_1[A_i] < a_1$, $\rho_1(t_1[A_i]) = b_{k1}$ when $t_1[A_i] > a_k$, and $\rho_1(t_1[A_i]) = b_{j1}$ when $a_j < t_1[A_i] < a_{j+1}$;
(c) for each A_i ($i \in [1,m]$), $\rho_2(t_2[A_i]) = \rho_1(t_1[A_i])$ when $t_1[A_i] = t_2[A_i]$; and, otherwise,
(d) $\rho_2(t_2[A_i]) = b_{02}$ when $t_2[A_i] < a_1$, $\rho_2(t_2[A_i]) = b_{k2}$ when $t_2[A_i] > a_k$, and $\rho_2(t_2[A_i]) = b_{j2}$ when $a_j < t_2[A_i] < a_{j+1}$.

Based on these, we give the details of the NP algorithm as follows:
(1) Guess two tuples t_1, t_2 of R such that $t_1[A_i], t_2[A_i] \in \mathsf{adom}(A_i)$ for each $i \in [1,m]$.
(2) Check whether $I = \{t_1, t_2\}$ satisfies Σ, but not φ. If so the algorithm returns "yes".

Both steps (1) and (2) can be done in PTIME in the size of $\Sigma \cup \{\varphi\}$, and it is easy to verify that $\Sigma \not\models \varphi$ if and only if the algorithm returns 'yes'. Hence the algorithm is in NP, and the problem is in coNP. □

Similar to the satisfiability analysis, it is known [FGJK08] that the implication analysis of CFDs is in PTIME when the CFDs are defined only with attributes that have an infinite domain. Analogous to Theorem 3.4.2, the result below shows that this is no

longer the case for CFD^{p}s, which does not find a counterpart in [FGJK08].

Theorem 3.4.6: *In the absence of finite domain attributes, the implication problem for* CFD^p*s remains co*NP*-complete.* □

Proof: The problem is in coNP following from Theorem 3.4.5.

We next show that the problem is coNP-hard by reduction from the 3SAT problem to the complement problem of the implication problem, where 3SAT is NP-complete (cf. Theorem 3.4.2). Given an instance ϕ of 3SAT, we construct a relational schema R, a set $\Sigma \cup \{\varphi\}$ of CFD^{p}s defined on R, such that the ϕ is satisfiable if and only if $\Sigma \not\models \varphi$.

The relational schema R and the set Σ of CFD^{p}s are the same as the corresponding ones in Theorem 3.4.2. Moreover, φ is defined as $(Z \to Z, T_p)$, where $T_p = \{(\_ \| \neq a)\}$. Intuitively, φ requires that for any tuple t of R, $t[Z] \neq a$. Along the same lines as Theorem 3.4.2, one can easily verify that the ϕ is satisfiable if and only if $\Sigma \not\models \varphi$. Thus the problem is coNP-hard. □

The implication analysis of CIND^{p}s. We next show that CIND^{p}s do not make their implication analysis harder. This is verified by extending the proof of their CINDs counterpart given in [BFM07] and Chapter 2.

Theorem 3.4.7: *The implication problem for* CIND^p*s is* EXPTIME*-complete.* □

Proof: The implication problem for CINDs is EXPTIME-hard [BFM07]. The lower bound carries over to CIND^{p}s, which subsume CINDs.

We next show that the problem is in EXPTIME by presenting an EXPTIME algorithm that, given a set $\Sigma \cup \{\psi\}$ of CIND^{p}s defined on a database schema $\mathcal{R}$, determines whether $\Sigma \not\models \psi$. Consider $\mathcal{R} = (R_1, \ldots, R_n)$ and $\psi = (R_a[X; X_p] \subseteq R_b[Y; Y_p], T_p)$. Moreover, for each attribute A appearing in $\mathcal{R}$, we define its active domain $\mathsf{adom}(A)$ based on $\Sigma \cup \{\psi\}$ along the same lines as the proof of Proposition 3.4.3.

In the following, we first show that $\Sigma \not\models \psi$ if and only if there exists a database D such that $D \models \Sigma$, $D \not\models \psi$, and D only consists of data values from those active domains. We then give the details of the EXPTIME algorithm, and finally we verify the correctness of the algorithm.

(1) We first show that $\Sigma \not\models \psi$ if and only if there exists a database D such that $D \models \Sigma$, $D \not\models \psi$ and D only consists of values from those active domains.

Assume first that there exists such a database D, then it is easy to see that $\Sigma \not\models \psi$.

Conversely, assume that $\Sigma \not\models \psi$. Then there must exist a database $D = (I_1, \ldots, I_n)$ such that $D \models \Sigma$ and $D \not\models \psi$. Moreover, there exists a tuple $t_a \in I_a$ such that $t_a[X_p] \asymp T_p[X_p]$, but there exists no tuple $t_b \in I_b$ such that $t_b[Y] = t_a[X]$ and $t_b[Y_p] \asymp T_p[Y_p]$.

We next show that we can construct another database $\rho(D)$ based on the database D, where ρ is a mapping that maps data values in D to data values in those active domains. The mapping ρ is defined as follows.

(a) We first define the mappings for the data values appearing in $\Sigma \cup \{\psi\}$. For each such data value c_1 appearing in D, $\rho(c_1) = c_1$.

(b) We then define the mappings for the data values appearing in the tuple t_a, but not in $\Sigma \cup \{\psi\}$. Consider an attribute A of the relational schema R_a with active domain $\mathsf{adom}(A) = C_A^0 \cup C_A^1$, where C_A^0 is the set of data values appearing in $\Sigma \cup \{\psi\}$. Assume *w.l.o.g.* that data values c_{min} and c_{max} are the smallest and the largest in C_A^0, respectively, and data values $c_1 < c_2$ are in C_A^0 such that there exists no data values $c \in C_A^0$ with $c_1 < c < c_2$.

When $t_a[A] < c_{min}$, $\rho(t_a[A])$ is equal to the smallest data value c in C_A^1 with $c < c_{min}$. When $t_a[A] > c_{max}$, $\rho(t_a[A])$ is equal to the smallest data value c in C_A^1 with $c > c_{max}$. And when $c_1 < t_a[A] < c_2$, $\rho(t_a[A])$ is equal to the smallest data value c in C_A^1 with $c_1 < c < c_2$.

(c) We finally define the mappings for all the other data values appearing neither in $\Sigma \cup \{\psi\}$ nor in the tuple t_a. Consider an attribute B of the relational schema R_i ($1 \leq i \leq n$) with active domain $\mathsf{adom}(B) = C_B^0 \cup C_B^1$, where C_B^0 is the set of data values appearing in $\Sigma \cup \{\psi\}$, and a tuple $t \in I_i$ in the database D. Similarly, assume *w.l.o.g.* that data values c_{min} and c_{max} are the smallest and the largest in C_B^0, respectively, and data values $c_1 < c_2$ are in C_B^0 such that there exists no data values $c \in C_B^0$ with $c_1 < c < c_2$.

When $t[B] < c_{min}$, $\rho(t[B])$ is equal to the largest data value c in C_B^1 such that $c < c_{min}$. When $t[B] > c_{max}$, $\rho(t[B])$ is equal to the largest data value c in C_B^1 with $c > c_{max}$. And when $c_1 < t[B] < c_2$, $\rho(t[B])$ is equal to the largest data value c in C_B^1 with $c_1 < c < c_2$.

Moreover, for each attribute $B \in Y$ with the corresponding attribute $A \in X$ in the CIND^p ψ, if (a) $t_a[A] \neq t[B]$, and (b) both $t_a[A]$ and $t[B]$ are in the same case mentioned above, *e.g.*, $t_a[A], t[B] < c_{min}$, we further require that $\rho(t[B]) \neq \rho(t_a[A])$.

From the CIND^p ψ and the construction of active domains, we know that $\mathsf{adom}(A) \subseteq \mathsf{adom}(B)$. Thus, it is always possible to find the mapping $\rho(t[B])$ since $t_a[A]$ and $t[B]$ already guarantee that there exist at least two distinct data values in $\mathsf{adom}(B)$ satisfying the above conditions.

It is easy to verify the following: (a) $\rho(D) \models \Sigma$, (b) $\rho(D) \not\models \psi$, and (c) $\rho(D)$ only consists of data values from those active domains.

(2) We then present the details of the algorithm.

The main idea is to maintain a directed edge-labeled bipartite graph $G(\Sigma, V, E, L)$

and a mapping $H(V)$. Here the Σ is the set of CIND^{p}s, the V consists of nodes in the form of '$\mathsf{R_i} : \mathsf{t_i}$', where R_i is a relational schema in $\mathcal{R}$, and t_i is an R_i tuple taking only values from the active domains defined before, and for each edge $e \in E$, $L(e) \in V$. Given a node u in V of G, $H(\mathsf{u})$ is the set of CIND^{p}s in Σ that are already applied to the node u in the process. With these two data structures, we can avoid *unnecessary* computations.

(a) It initializes the node set $V = \{\mathsf{N_{root}}\}$, the edge set $E = \emptyset$, and $H(\mathsf{N_{root}}) = \emptyset$. Here the node $\mathsf{N_{root}}$ = '$\mathsf{R_a} : \mathsf{t_a}$', where t_a is an R_a tuple such that for each attribute A of the relational schema R_a, the data value $t_a[A]$ is drawn from $\mathsf{adom}(A)$.

(b) For each node u = '$\mathsf{R_i} : \mathsf{t_i}$' in V, it checks whether there exists a CIND^p $\psi' = (R_i[U;U_p] \subseteq R_j[V;V_p], T_{p_{\psi'}})$ in Σ such that $t_i[U_p] \asymp T_{p_{\psi'}}[U_p]$.

(c) If there exists such a CIND^p ψ' for the node u = '$\mathsf{R_i} : \mathsf{t_i}$', it first generates a set $S_{(u,\psi')}$ of new nodes, and then updates the graph G and the mapping H accordingly.

The set $S_{(u,\psi')}$ consists of all nodes in the form of '$R_j : t_j$', where t_j is an R_j tuple such that $t_j[V] = t_i[U]$, $t_j[V_p] \asymp T_{p_{\psi'}}[V_p]$, and the data value $t_j[B]$ is drawn from $\mathsf{adom}(B)$ for all attributes B of the relational schema R_j.

Then, for the mapping H, $H(\mathsf{u}) = H(\mathsf{u}) \cup \{\psi'\}$, and for each node u' in $S_{(u,\psi')} \setminus V$, $H(\mathsf{u}) = \emptyset$. For the graph $G(\Sigma, V, E)$, its node set V is updated to be $V \cup S_{(u,\psi')}$, and its edge set E is updated as follows:

- add an edge (u, ψ') with label u to E; and
- for each node u' in $S_{(\mathsf{u},\psi')}$, it simply adds an edge (ψ', u') with label u to E.

(d) The above process repeats until there are no more changes for the node set V and the edge set E of the graph $G(\Sigma, V, E, L)$.

(e) The algorithm finally checks whether $\Sigma \not\models \psi$ based on the graph G.

- Let S_b be the set of nodes '$R_b : t_b$' in V such that $t_b[Y] = t_a[X]$ and $t_b[Y_p] \asymp T_p[Y_p]$.
- It then recursively enlarges the set S_b to include those nodes u in Σ and V such that (a) for the node u is in V, there exists a node ψ' in Σ with an edge (u, ψ') in Σ, and ψ' is in S_b; and (b) for the node u in Σ, there exists a label such that the neighboring nodes u' of node u with the same label all appear in S_b.
- After the set S_b reaches a fixpoint, if the node $\mathsf{N_{root}}$ = '$\mathsf{R_a} : \mathsf{t_a}$' is not in S_b, the algorithm simply returns 'yes'; and, otherwise, it starts the entire process with a distinct R_a tuple.

(f) The algorithm returns 'no' if for all R_a tuples, step (e) does not return a 'yes'.

(3) We finally verify the correctness of the algorithm. We show that $\Sigma \not\models \psi$ if and only if the algorithm returns 'yes'.

First assume that the algorithm returns 'yes' at step (e) for a node $\mathsf{N}_{\mathsf{root}} = `R_a : t_a`$'. By collecting all the tuples in the node set V of the graph, but not in the S_b at step (e), we have a nonempty database instance D. It is easy to verify that $D \models \Sigma$, but not $D \models \psi$. From this, it follows that $\Sigma \not\models \psi$.

Conversely, assume that $\Sigma \not\models \psi$. By (1), there exists a nonempty database instance $D = \{I_1, \ldots, I_n\}$ such that $D \models \Sigma$, but not $D \models \psi$, and D only consists of data values drawn from those active domains. That is, there exists an R_a tuple $t_a \in I_a$ such that $t_a \asymp T_p[X_p]$, and there exists no R_b tuple $t_b \in I_b$ such that $t_b[Y] = t_a[X]$ and $t_b[Y_p] \asymp T_p[Y_p]$. Assume *w.l.o.g.* that the number of tuples in the database D is minimum.

Consider the graph G at step (e) with the initial node $\mathsf{N}_{\mathsf{root}} = `R_a : t_a`$'. Let S_D be the set of nodes $\mathsf{u} = `R_i : t_i`$' in the node set V such that tuple $t_i \in I_i$ of the database D. Since the database D is minimum, all nodes in S_D must also appear in the graph G. Since $D \models \Sigma$, but not $D \models \psi$, all nodes u in S_D are not in the set S_b, *i.e.,* $S_D \cap S_b$ is empty. Thus, the algorithm returns 'yes' at step (e) since the node $\mathsf{N}_{\mathsf{root}} = `R_a : t_a`$' is not in S_b.

(4) To see that the algorithm is in exponential time, observe the following. (a) The number of nodes in the graph G is bounded by $|I| + |\Sigma|$, where $|I|$ is the maximum size $|I|$ of a database instance and $|\Sigma|$ is the number of CIND^{p}s in Σ. Therefore, the size of G is bounded by $O(2^{n^2})$ as argued in the proof of Proposition 2.3.9, where n is the size of the input here. (b) It takes $O(2^{n^2})$ time to find a node u in V and a CIND^p ψ' in Σ with $\psi' \notin H(\mathsf{u})$. (c) The number of nodes in $S_{u,\psi}$ is bounded by $O(2^{n^2})$. (d) The number of nodes in S_b is is bounded by the number of nodes in G, *i.e.,* $O(2^{n^2})$. (e) It takes $O(2^{n^2} * 2^{n^2} * 2^{n^2})$ time to reach the fixpoint for the set S_b. (f) Finally, there are at most $O(2^{n^2})$ number of R_a tuples to be tested.

Based on these one can readily verify that the algorithm is indeed in EXPTIME. □

It is known [BFM07] that the implication problem is PSPACE-complete for CINDs defined with infinite-domain attributes. Similar to Theorem 3.4.6, below we present a new result showing that this no longer holds for CIND^{p}s.

Theorem 3.4.8: *In the absence of finite domain attributes, the implication problem for* CIND^p*s remains* EXPTIME*-complete.* □

Proof: The problem is in EXPTIME following from Theorem 3.4.7. We next show that the problem is EXPTIME-hard by reduction from the implication problem for CINDs in the general setting, which is EXPTIME-complete (cf. [BFM07]). Given a set $\Sigma \cup \{\psi\}$ of

CINDs defined on a database schema $\mathcal{R} = (R_1, \ldots, R_n)$, we construct a database schema $\mathcal{R}' = (R'_1, \ldots, R'_n)$, where the relational schema R'_i ($i \in [1,n]$) consists of infinite-domain attributes only, and a set $\Sigma' \cup \{\psi'\}$ of CIND^{p}s on $\mathcal{R}'$. And, moreover, $\Sigma \models \psi$ iff $\Sigma' \models \psi'$.

We start with constructing $\mathcal{R}'$. For each $R_i(A_1, \ldots, A_k)$ of $\mathcal{R}$, we define $R'_i(A'_1, \ldots, A'_k)$ such that for each attribute A'_j ($j \in [1,k]$), $\mathsf{dom}(A'_j) = \mathsf{dom}(A_j)$ if $\mathsf{dom}(A_j)$ is infinite and $\mathsf{dom}(A'_j)$ is integer, a totally ordered infinite domain. Moreover, we define a mapping $\rho_{i,j}$ that maps data values for the finite domain $\mathsf{dom}(A_j) = \{a_1, \ldots, a_h\}$ to an infinite integer domain: (1) randomly choose h consecutive integers $\{b_1, \ldots, b_h\}$ such that for each $i \in [1, h-1]$, $b_{i+1} = b_i + 1$; and (b) we define $\rho_{i,j}(a_i) = b_i$ for each $i \in [1,h]$. We further introduce two extra integers $b_0 = b_1 - 1$ and $b_{h+1} = b_h + 1$, denoted as $\rho_{i,j}.b_0$ and $\rho_{i,j}.b_{h+1}$, respectively. Note that this is always doable. For the ease of description, we also denote $\rho_{i,j}$ as ρ when it is clear from the context.

We next define the set $\Sigma' \cup \{\psi'\}$ of CIND^{p}s on $\mathcal{R}'$ based on the mappings defined above. For each CIND $\varphi = (R_i[U; U_p] \subseteq R_j[V; V_p], t_p)$ in $\Sigma \cup \{\psi\}$, we define a CIND^p $\varphi' = (R'_i[U'; U'_p, X'_p] \subseteq R'_j[V'; V'_p, Y'_p], T_p)$, where (1) U' (resp. V', U'_p, V'_p) corresponds to U (resp. V, U_p, V_p); (2) X'_p (resp. Y'_p) corresponds to those finite-domain attributes in R'_i (resp. R'_j), but not in U (resp. V); and (3) $T_p = \{t_{p1}, t_{p2}, t_{p3}\}$ such that for each attribute A' in U'_p or V'_p, (a) $t_{p1}[A']$ is '$= t_p[A]$' and $t_{p2}[A'] = t_{p3}[A'] =$ '_' if $\mathsf{dom}(A)$ is infinite, and (b) $t_{p1}[A']$ is '$= \rho(t_p[A])$' and $t_{p2}[A'] = t_{p3}[A'] =$ '_' if $\mathsf{dom}(A)$ is finite; and (c) for all the remaining attributes B' in X'_p or Y'_p, $t_{p1}[B'] =$ '_', $t_{p2}[B'] =$ '$> \rho.b_0$', and $t_{p3}[B'] =$ '$< \rho.b_{h+1}$'.

Finally, one can easily verify that $\Sigma \models \psi$ iff $\Sigma' \models \psi'$. Following from this, the problem is EXPTIME-hard. □

The implication analysis of CFD^{p}s and CIND^{p}s. When CFD^{p}s and CIND^{p}s are taken together, their implication analysis is beyond reach in practice. This is not surprising since the implication problem for FDs and INDs is already undecidable [AHV95]. Since CFD^{p}s and CIND^{p}s subsume FDs and INDs, respectively, from the undecidability result for FDs and INDs, the corollary below follows immediately.

Corollary 3.4.9: *The implication problem for* CFD^p*s and* CIND^p*s is undecidable.* □

3.5 Related Work

To our knowledge, no previous work has studied extensions of CFDs to capture disjunctions and negations, cardinality constraints and synonym rules, or built-in predicates ($\neq, <, \leq, >, \geq$), and extensions of CINDs to capture built-in predicates ($\neq, <, \leq, >, \geq$).

Constraint-based data cleaning was introduced in [ABC03b], which proposed to use dependencies, *e.g.,* FDs, inclusion dependencies (INDs) and denial constraints, to detect errors in real-life data (see, *e.g.,* [Cho07] for a comprehensive survey). As an extension of traditional FDs, CFDs were developed in [FGJK08], which showed that the satisfiability problem and implication problem for CFDs are NP-complete and coNP-complete, respectively. Along the same lines, CINDs were proposed in [BFM07] to extend INDs. It was shown [BFM07] that the satisfiability and implication problems for CINDs are in constant time and EXPTIME-complete, respectively. There have been extensions of CFDs to support ranges of values in pattern tuples [GKK$^+$08], which can be treated as a special case of CFD^{p}s without inequalities ($\neq$).

Close to our work are dependencies of [BCW99, BP83, Mah97, MS96] developed for constraint databases. Constraints of [BP83], also referred to as conditional functional dependencies, are of the form $(X \to Y) \to (Z \to W)$, where $X \to Y$ and $Z \to W$ are standard FDs. Constrained dependencies of [Mah97] extend [BP83] by allowing $\xi \to (Z \to W)$, where ξ is an arbitrary constraint that is not necessarily an FD. In a nutshell, these dependencies are to apply FD $Z \to W$ only to the subset of a relation that satisfies $X \to Y$ or ξ. They cannot express even CFDs since $Z \to W$ does not allow patterns with constants as found in CFDs, eCFDs, CFD^{c}s, and CFD^{p}s. More expressive are constraint-generating dependencies (CGDs) of [BCW99] and constrained tuple-generating dependencies (CTGDs) of [MS96], of the form $\forall\bar{x}(R_1(\bar{x}) \wedge \ldots \wedge R_k(\bar{x}) \wedge \xi(\bar{x}) \to \xi'(\bar{x}))$ and $\forall\bar{x}(R_1(\bar{x}) \wedge \ldots R_k(\bar{x}) \wedge \xi \to \exists\bar{y}(R'_1(\bar{x},\bar{y}) \wedge \ldots \wedge R'_s(\bar{x},\bar{y}) \wedge \xi'(\bar{x},\bar{y}))$, respectively, where R_i, R'_j are relation symbols, and ξ, ξ' are arbitrary constraints. While CGDs can express CFDs, eCFDs and CFD^{p}s, and CTGDs can express CFDs, eCFDs, CFD^{p}s and CIND^{p}s, neither of them can capture CFD^{c}s which support cardinality constraints and synonym rules. Built-in predicates and arbitrary constraints are supported by CGDs, however, their satisfiability and implication problems are not studied in the presence of finite-domain attributes. The work of CFD^{p}s also provides lower bounds for the satisfiability and implication analysis of CGDs, by using patterns with built-in predicates only. The increased expressive power of CTGDs comes at the price of a higher complexity: both their satisfiability and implication problems are undecidable.

In addition, we are not aware of any applications of these constraints in data cleaning. A detailed discussion about the differences between CFDs and these extensions ([BCW99, BP83, Mah97, MS96]) also appears in [FGJK08].

Synonym rules have been studied for record matching [ACG02, ACK08] in the form of transformation rules. However, no previous work has studied how to express these in dependencies, or their impact on the static analyses of dependencies.

Cardinality constraints have been studied for relational data [Kan80] to constrain the domains of attributes, and for object-oriented databases to restrict the extents of classes [CL94]. Numerical dependencies [GM85], which generalize FDs with cardinality constraints, have also been proposed for schema design. These constraints differ from CFD^{c}s in that they cannot constrain tuples with a pattern specified in terms of constants. Query answering has been investigated for aggregate queries, FDs and denial constraints [ABC$^+$03a, BBFL08], which differ from this work in that neither these dependencies can express cardinality constraints, nor the impact of cardinality constraints on the satisfiability and implication analyses has been considered.

Codd tables, variable tables and conditional tables have been studied for incomplete information [IL84, Gra91], which also allow both variables and constants in the specifications. As clarified in [FGJK08], these formalisms differ from CFDs, eCFDs, CFD^{c}s, and CFD^{p}s, in that each of these tables is used as a representation of possibly infinitely many relation instances, one instance for each instantiation of variables in the table. No instance represented by these table formalisms can include two tuples that result from different instantiations of a table tuple. In contrast, all pattern tuples in a pattern tableau of a CFD, eCFD, CFD^c, or CFD^p constrain a *single* relation instance, which can contain any number of tuples that are all instantiations of unnamed variables in the same pattern tuple.

Chapter 4

Detecting Data Inconsistencies

We have proposed (a) conductional inclusion dependencies (CINDs) which is an extension of inclusion dependencies (INDs, [AHV95]) in Chapter 2, (b) eCFDs, CFD^{c}s and CFD^{p}s which are extensions of conditional functional dependencies (CFDs, [FGJK08]) in Chapter 3, and (c) CIND^{p}s which is an extension of CINDs in Chapter 3. It was shown that these new classes of data dependencies could capture data inconsistencies commonly found in practice, which is beyond the expressive power of both CFDs and traditional dependencies such as FDs and INDs [AHV95].

In this chapter, we focus on how to detect data inconsistencies based on these dependencies, *i.e.,* the *error detection problem*. The error detection problem is to find, given a set Σ of data dependencies (*e.g.,* CINDs, eCFDs, CFD^{c}s, CFD^{p}s or CIND^{p}s) defined on a database schema $\mathcal{R} = (R_1, \ldots, R_n)$, and a database instance $D = (I_1, \ldots, I_n)$ of $\mathcal{R}$ as input, the maximum subset $(I'_1, \ldots, I'_n)$ of D such that for each $i \in [1, n]$, $I'_i \subseteq I_i$ and each tuple in I'_i violates at least one data dependency in Σ. We denote the set as $\mathsf{vio}(D, \Sigma)$, referred to it as *the violation set* of D *w.r.t.* Σ. Note that for CFDs, eCFDs, CFD^{c}s and CFD^{p}s, it suffices to consider a single relation since these dependencies are defined on a single relation. This does not lose generality since we can handle relations one by one when there are multiple relations.

We have developed SQL-based techniques, which significantly extend the one used for CFDs [FGJK08], and can be easily implemented on the top of current DBMS. The main idea to encode data dependencies with data tables, and, thus, both data and constraints are treated uniformly and stored in a DBMS.

Organizations. We explain how to encode eCFDs (resp. CFD^{c}s, and CFD^{p}s and CIND^{p}s) by using data tables, and how to detect data inconsistencies with eCFDs (resp. CFD^{c}s, and CFD^{p}s and CIND^{p}s) in Section 4.1 (resp. Section 4.2 and Section 4.3).

4.1 Detecting Inconsistencies with eCFDs

In this section, we develop SQL techniques for detecting violations *w.r.t.* eCFDs.

We consider *static* and *dynamic* settings, stated as follows:

Static Setting. Given a set Σ of eCFDs defined on a database schema $\mathcal{R}$ with a single relation, and a database D of $\mathcal{R}$, a *batch detection algorithm* is to find the *violation set* $\mathsf{vio}(D)$ *w.r.t.* Σ, *i.e.,* the set of all tuples in D that violate some eCFDs in Σ.

Dynamic Setting. Given the D and the Σ as above, the violation set $\mathsf{vio}(D)$ of D *w.r.t.* Σ, and updates ΔD to the database D, an *incremental detection algorithm* is to find the set $\Delta\mathsf{vio}(D)$ such that $\Delta\mathsf{vio}(D) \oplus \mathsf{vio}(D)$ is the violation set $\mathsf{vio}(\Delta D \oplus D)$ *w.r.t.* Σ, where $\Delta S \oplus S$ denotes applying the updates ΔS to the set S. Here the updates ΔD can be either a set of tuple *insertions* or *deletions*, denoted by ΔD^+ and ΔD^-, respectively.

In Sections 4.1.2 and 4.1.3, we develop a batch algorithm and an incremental detection algorithm, referred to as BATCHDETECT and INCDETECT, respectively. Both algorithms only generate SQL queries to detect violations. This is important since eCFD violation detection can then be directly implemented on top of DBMS, and we can therefore benefit from existing optimization techniques of DBMS. Better still, in both settings, only a *fixed* number of SQL queries are needed, no matter how many eCFDs are in Σ, how many pattern tuples are in the eCFDs, and how large the sets are in each pattern-tuple attribute. The key idea is to treat pattern tableaux in Σ as *data tables*, rather than as *meta-data*.

Before we present our detection algorithms we decide on a uniform way of representing the set of violations. Instead of simply returning the tuples in D that violate some eCFD in Σ, we *explicitly store* whether a tuple in D is a violation or not. More precisely, we extend the schema R of D with two Boolean attributes: SV (for "Single tuple Violation") and MV (for "Multiple tuple Violation"). That is, $t.\mathsf{SV} = 1$ if t violates an eCFD in Σ all by itself; and $t.\mathsf{SV} = 0$ otherwise. Similarly, $t.\mathsf{MV} = 1$ if t violates an embedded FD for an eCFD in Σ; and $t.\mathsf{MV} = 0$ otherwise. Hence, $t \in \mathsf{vio}(D)$ if either $t.\mathsf{SV} = 1$ or $t.\mathsf{MV} = 1$.

4.1.1 Encoding eCFDs with Data Tables

To achieve this, we encode Σ with data tables. This encoding is used both in the batch detection algorithm and in the incremental detection algorithm, and therefore we explain it in detail.

We start by encoding the set Σ of eCFDs with a data table enc, a data table enc_{A_L}

(1) enc

cid	CT_L	AC_R
1	2	3
2	1	1
3	1	-1

(2) T_{CT_L}

cid	CT_L
1	NYC
1	LI
2	Albany
2	Troy
2	Colonie
3	NYC

(3) T_{AC_R}

cid	AC_R
2	518
3	212
3	718
3	646
3	347
3	917

Figure 4.1: Encoding of eCFDs

for each attribute A appearing in the LHS of eCFDs in Σ, and a data table enc_{B_R} for each attribute B appearing in the RHS of eCFDs in Σ.

The enc consists of (a) a unique identifier cid for each eCFD in Σ, and (b) an attribute A_L (resp. A_R) for each attribute A appearing in the LHS (resp. RHS) of some eCFDs in Σ. In total, the number of attributes in enc is bounded by $2|R|+1$, where $|R|$ is the arity of R. All the other data tables enc_{A_L} or enc_{B_R} are binary, which consists of an attribute cid, and the attribute A_L or B_R. Note that the number of data tables to encode all eCFDs in Σ is bounded by $2|R|+1$ as well.

For each eCFD $\varphi = (R : X \to Y,\ Y_p,\ T_p)$ in Σ, we generate a distinct cid $\mathsf{id}(\varphi, t_p)$ for each $t_p \in T_p$, and do the following.

- Add a tuple t_1 to enc such that (a) $t_1[\mathsf{cid}] = \mathsf{id}(\varphi, t_p)$; (b) for each attribute $A \in X$, $t_1[A_L] = 1$ (resp. $t_1[A_L] = 2$ and $t_1[A_L] = 3$) if $t_p[A_L] = S$ (resp. $t_p[A_L] = \overline{S}$ and $t_p[A_L]$ = '_'); (c) for each attribute $B \in Y$, $t_1[B_R] = 1$ (resp. $t_1[B_R] = 2$ and $t_1[B_R] = 3$) if $t_p[B_R] = S$ (resp. $t_p[B_R] = \overline{S}$ and $t_p[B_R]$ = '_'); (d) for each attribute $C \in Y_p$, $t_1[C_R] = -1$ (resp. $t_1[C_R] = -2$ and $t_1[C_R] = -3$) if $t_p[C_R] = S$, (resp. $t_p[C_R] = \overline{S}$ and $t_p[C_R]$ = '_'); and (c) $t_1[B]$ = 0 for all other attributes B_L in enc.
- For each attribute $A \in X$ such that $t_p[A_L] = S$ or $t_p[A_L] = \overline{S}$, add a tuple t_a to enc_{A_L} for each element $a \in S$ with $t_a[\mathsf{cid}] = \mathsf{id}(\varphi, t_p)$ and $t_a[\mathsf{A_L}]$ = 'a'.
- Similarly, for each attribute $B \in Y \cup Y_p$ such that $t_p[B_R] = S$ or $t_p[B_R] = \overline{S}$, add a tuple t_b to enc_{B_R} for each element $b \in S$ with $t_b[\mathsf{cid}] = \mathsf{id}(\varphi, t_p)$ and $t_b[\mathsf{B_R}]$ = 'b'.

Example 4.1.1: Consider the eCFDs φ_1 and φ_2 defined on the relational schema cust in Example 3.2.2 of Section 3.2.1 in Chapter 3, where

φ_1: $(\mathsf{CT} \to \mathsf{AC}, \emptyset, \{t_p^0, t_p^1\})$, where $t_p^0[\mathsf{CT}] = \overline{\{\text{NYC, LI}\}}$ and $t_p^0[\mathsf{AC}]$ ='_', and
$t_p^1[\mathsf{CT}] = \{\text{Albany, Troy, Colonie}\}$ and $t_p^1[\mathsf{AC}] = \{518\}$;
φ_2: $(\mathsf{CT} \to \emptyset, \mathsf{AC},\ \{t_p^2\})$, where $t_p^2[CT] = \{\text{NYC}\}$ and $t_p^2[AC] = \{212, 718, 646, 347, 917\}$.

We illustrate the encoding of φ_1 and φ_2 in Fig. 4.1 with three data tables enc, enc_{CT_L} and enc_{AC_R}. Note that enc encodes *all* eCFDs in Σ uniformly, one tuple for each pattern tuple in those eCFDs. □

4.1.2 An SQL-based Batch Algorithm

We first consider the static case and outline algorithm BATCHDETECT. Here we extend the approach proposed in [FGJK08] and generate a pair of SQL queries for violation detection and corresponding update statements:

- Query Q_{sv} finds all single-tuple violations due to violations of the pattern constraints enforced by eCFDs in Σ;
- Query Q_{mv} identifies multiple-tuple violations caused by a violation of an FD embedded in some eCFD in Σ.
- Given these two queries, BATCHDETECT performs update statements to D and sets the SV (resp. MV) attribute to 1 for those tuples returned by Q_{sv} (resp. Q_{mv}).

Algorithm BATCHDETECT. The following SQL queries are employed by BATCHDETECT.

(1) We first detect single-tuple violations that are caused by pattern constraints, *i.e.,* tuples in D that satisfy the pattern constraints of the LHS of an eCFD in Σ but do not satisfy those of its RHS. The encoding is similar to the one for CFDs presented in [FGJK08], by literally expressing pattern-constraint violation in SQL. In contrast to CFDs, patterns are now sets (or the complement thereof). For this, we need to express the fact that an element is in (resp. not in) a set by means of `EXISTS` (resp. `NOT EXISTS`). Figure 4.2 shows the query Q_{sv} for the eCFDs given in Example 4.1.1

(2) We next detect the multiple-tuple violations that are caused by violations of the embedded FDs in the eCFDs in Σ. Similar to [FGJK08], detection of such violations can be readily expressed using `GROUP BY` in SQL. However, we have to group by different attributes depending on the eCFD under consideration. This can be achieved by blanking out (using a constant "@" not appearing in any database) those attributes that are not relevant. Attributes irrelevant to the embedded FD have non-positive entries in the relation enc . We use the `CASE` construct in the `SELECT` statement to replace the attributes values of tuples in D by '@' if the attribute is irrelevant to the embedded FD; otherwise we return the attribute value of the tuple instead. We provide an example query Q_{mv} in Fig. 4.3. Note that Q_{mv} returns tuples of the form (cid, p), where cid

select cust.$*$
from cust, enc
where (enc.$\mathsf{CT_L} \neq 1$ **or** (enc.$\mathsf{CT_L} = 1$ **and exists** $Q^{\mathsf{CT_L}}$))
  and (enc.$\mathsf{CT_L} \neq 2$ **or** (enc.$\mathsf{CT_L} = 2$ **and not exists** $Q^{\mathsf{CT_L}}$))
  and ((**abs**(enc.$\mathsf{AC_R}$) $= 1$ **and not exists** $Q^{\mathsf{AC_R}}$) **or** (**abs**(enc.$\mathsf{AC_R}$) $= 2$ **and exists** $Q^{\mathsf{AC_R}}$)),

where (a) **abs** is a function which returns the absolute value of an integer, and (b) for all attributes $B = A_L$ or $B = A_R$, Q^B stands for

select $*$
from enc$_B$
where enc.cid = enc$_B$.cid **and** cust.A = enc$_B$.B

Figure 4.2: The query Q_{sv}

select m.cid, m.$\mathsf{CT_L}$, m., **count(*)**
from macro m
group by m.cid, m.$\mathsf{CT_L}$
having count(*1) > 1,

where (a) macro stands for

select distinct enc.cid **as** cid,
    (**case** enc.$\mathsf{CT_L}$ **when** > 0 **then** cust.CT **else** '@' **end**) **as** $\mathsf{CT_L}$,
    (**case** enc.$\mathsf{AC_R}$ **when** > 0 **then** cust.AC **else** '@' **end**) **as** $\mathsf{AC_R}$
from cust, enc
where (enc.$\mathsf{CT_L} \neq 1$ **or** (enc.$\mathsf{CT_L} = 1$ **and exists** $Q^{\mathsf{CT_L}}$))
  and (enc.$\mathsf{CT_L} \neq 2$ **or** (enc.$\mathsf{CT_L} = 2$ **and not exists** $Q^{\mathsf{CT_L}}$))

Figure 4.3: The query Q_{mv}

is an identifier for an eCFD (as given by enc) and p is a tuple consisting of constant values and "@"s. Intuitively, if a tuple $t \in D$ matches p for some $(cid, p) \in Q_{mv}(D)$ then it violates the embedded FD of the eCFD identified by cid.

(3) We set the SV attribute to "1" for tuples returned by Q_{sv}. For the MV-attribute, note that a tuple t in D is involved in a multiple tuple violation iff there exists a $(cid, p) \in Q_{mv}(D)$ such that t matches p. An additional SQL query identifies these tuples and updates their MV-attribute to "1".

Putting these together, given schema R and set Σ of eCFDs defined on R, algorithm BATCHDETECT generates SQL queries and update statements for detecting pattern-constraint violations and embedded FD violations, respectively, by capitalizing on the encoding given above.

Remarks. (1) The schema of the encoding relations, namely, enc and the binary relations T_A, is determined by the schema R rather than Σ. (2) The entire encoding relations are *linear* in the size of the input eCFDs Σ. (3) The detection SQL queries conduct two passes of the database D, regardless of the number of eCFDs and the size of pattern tuples in Σ. That is, they have *the same data complexity as* detection queries for CFDs [FGJK08]. Note that these queries necessarily use `EXISTS` and `NOT EXISTS` ; but these operations are only applied to auxiliary relations that encode the sets of constants mentioned in the eCFD patterns, rather than to the underlying database. Indeed, for each data tuple Q_{sv} conducts a linear scan of Σ, *the same as* its CFD counterpart; similarly for Q_{mv}. It is also worth remarking that the coding of eCFDs for algorithm BATCHDETECT is more involved than that of [FGJK08], in order to cope with the *set* elements in pattern tuples. A direct extension of the technique of [FGJK08] may lead to excessive space overhead, as opposed to the linear space taken by BATCHDETECT.

4.1.3 An SQL-based Incremental Algorithm

We next present incremental algorithm INCDETECT in response to database updates ΔD. Of course, BATCHDETECT can be directly applied to the new database obtained by updating the database D with ΔD. We want to *incrementally* detect violations because the deletion or insertion of a small number of tuples only affects a small part of D and as a result, one only needs to identify violations in the affected part rather than inspect the *entire* database. Algorithm INCDETECT aims to minimize unnecessary recomputation conducted for finding violations.

Like BATCHDETECT, Algorithm INCDETECT also generates SQL queries to identify changes to the violations of pattern constraints and changes to the violations of embedded FDs in Σ. In addition, it maintains an auxiliary relation in order to reuse previous computations. Observe that tuple deletions ΔD^- may remove violations from D but do not introduce new violations; on the other hand, tuples insertions ΔD^+ may add new violations introduced by inserted tuples alone or together with tuples in D.

Auxiliary relation. We maintain an auxiliary relation $\mathsf{Aux}(D)$, initialized by storing the query result of Q_{mv} from BATCHDETECT on D. The relation $\mathsf{Aux}(D)$ consists of tuples of the form (cid, p) where cid is an eCFD identifier and p is a tuple consisting of constants and "@". As noted above, each (cid, p) corresponds to the set of tuples that are involved in a multiple-tuple violation of the eCFD identified by cid and that match

p. We next describe how $\mathsf{Aux}(D)$ is maintained during updates on D and how it can be used to incrementally compute the updated set of violations.

Algorithm INCDETECT**.** Initially, we are given (i) D in which the SV and MV attributes correctly indicate the violations of Σ (this can be obtained by running algorithm BATCHDETECT); (ii) the set of updates ΔD; and (iii) the auxiliary relation $\mathsf{Aux}(D)$ (initialized as described above).

Algorithm INCDETECT needs to perform several tasks: it needs to compute $D \oplus \Delta D$, correctly update the SV and MV attributes for the tuples in $D \oplus \Delta D$, and update $\mathsf{Aux}(D)$ to $\mathsf{Aux}(D \oplus \Delta D)$. Moreover, INCDETECT performs these tasks using SQL statements only. Since deletions and insertions are dealt with in different ways, we treat them separately. We first consider the case of deletions.

Tuple deletions. Let ΔD^- be the set of tuples that are to be deleted from D. We first explain how $\mathsf{Aux}(D)$ is updated and then show how it is used to correctly update the multiple violation attribute MV in D. Because deletions do not eliminate single tuple violations (except for those that are in ΔD^-), we do not need to update the SV attribute.

(1) To update $\mathsf{Aux}(D)$, observe the following: a tuple (cid, p) can be removed from $\mathsf{Aux}(D)$ if it either does not match any tuple in $D \oplus \Delta D^-$, or all matching tuples in $D \oplus \Delta D^-$ do not violate the embedded FD of the eCFD identified by cid. It suffices to only consider (cid, p)'s that are *potentially* affected by the update ΔD^-, *i.e.,* those (cid, p)'s that match a tuple in ΔD^-, and thus avoid unnecessary computations After removing these (cid, p)'s from $\mathsf{Aux}(D)$, we obtain the updated $\mathsf{Aux}(D \oplus D^-)$.

(2) In order to update the MV attribute, we first observe that it is sufficient to only consider tuples t in D with $t.\mathsf{MV} = 1$. For each such t, we check whether it does not match any p in $(cid, p) \in \mathsf{Aux}(D \oplus \Delta D^-)$, and if so, update $t.\mathsf{MV}$ to 0.

Tuple insertions. Let $\Delta\mathsf{D}^+$ be the set of tuples to be inserted into D. We perform the following steps:

(1) We first detect the single-tuple violations in ΔD^+. That is, we apply Q_{sv} of BATCHDETECT on ΔD^+ and update the SV-attribute in ΔD^+ accordingly.

(2) Next, we identify new multiple-tuple violations in $D \oplus \Delta D^+$ by performing the following steps:

(2.a) Update the MV attribute of tuples in ΔD^+ that violate an eCFD together with a tuple in D. These can be easily found by matching tuples in $\mathsf{Aux}(D)$ with tuples in ΔD^+.

(2.b) Update Aux(D). Denote by D_{clean} the set of tuples in D satisfying Σ, which can be easily identified using the MV attribute. We insert tuples (cid, p) into Aux(D) that correspond to violations between (previously clean) tuples in D and tuples in ΔD^+.
(2.c) We then update the MV attribute for tuples in $D_{\mathsf{clean}} \oplus \Delta D^+$ that match some tuple (cid, p) in Aux(D).
(2.d) To account for multiple-tuple violations caused by tuples in ΔD^+ alone, we have to update Aux(D) again. For this, we run Q_{mv} on ΔD^+ and insert the result tuples into Aux(D). After this step, Aux(D) becomes Aux($D \oplus \Delta D^+$).
(2.e) Finally, we add ΔD^+ to D and update the MV-attribute of tuples in ΔD^+ that match a (cid, p) tuple in Aux($D \oplus \Delta D^+$).

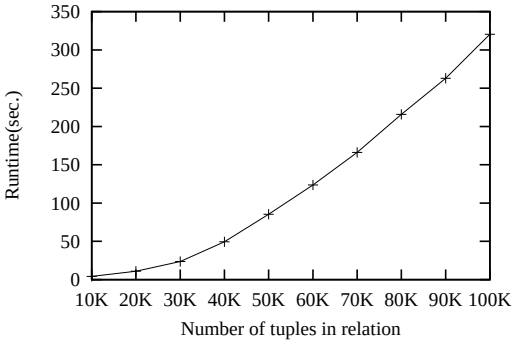
It is easily verified that the above steps correctly maintain both the auxiliary relation and violation set for both tuple deletions and insertions. Moreover, they can all be performed using SQL statements.

Remarks. (1) Algorithm IncDetect uniformly employs an auxiliary relation and SQL queries to handle *multiple* tuple deletions and insertions, for the *entire* set Σ of eCFDs. This is the first SQL-based technique for incrementally detecting violations of *multiple* eCFDs. (2) Recomputation is avoided by only considering relevant tuples in D using both the auxiliary relation and the update set.

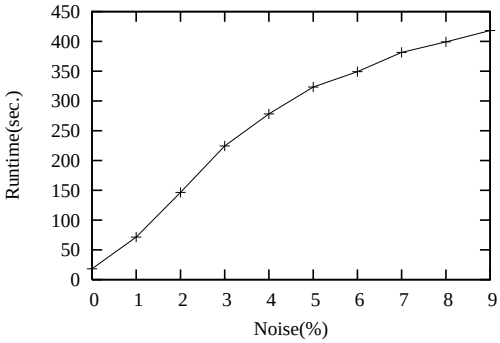
4.1.4 Experimental Study

Our experimental study focuses on the SQL-based algorithms BatchDetect and IncDetect for detecting data inconsistencies. We evaluate (1) the scalability of BatchDetect and IncDetect *w.r.t.* the size of databases, the complexity of eCFDs and the error rate in the databases, and (2) the performance of IncDetect versus BatchDetect in response to database updates.

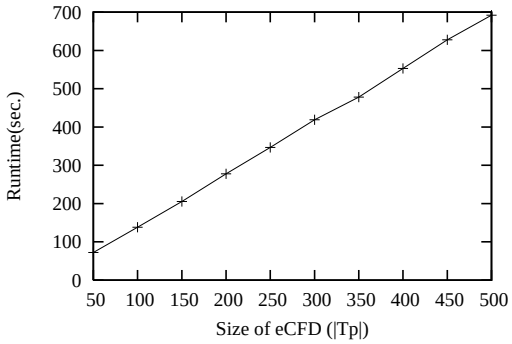
Experimental setting. Our experiments are based on an extension of the cust relation shown in Fig. 3.1 in Chapter 3, that adds information about items bought by different customers. We scraped real-life CT, AC, ZIP data for cities and towns in the US and different items, such as books, CDs and DVDs, from online stores. Using this, we wrote a program to generate synthetic datasets, denoted by D. We considered two parameters of the datasets D: |D| for the number of tuples in D, ranging from 10k to 100k, and noise% for the percentage of tuples in D that were modified to violate an eCFD, ranging from 0% to 9%. The modification consists of changing tuples in D in attributes in the right-hand side of some eCFDs from a correct to an incorrect value.



(a) Scalability in |D|



(b) Scalability in noise



(c) Scalability in number of constraints

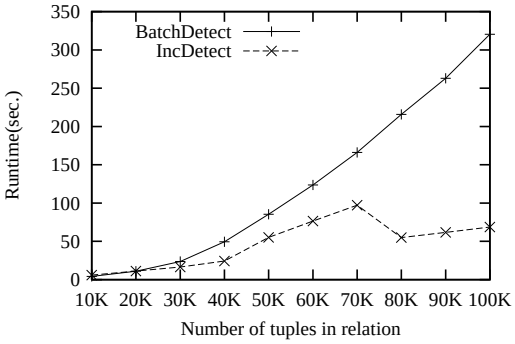
Figure 4.4: BATCHDETECT

We used a set Σ consisting of 10 eCFDs to express real-life semantics of the real-life data, including the two eCFDs of Fig. 3.2 in Chapter 3. We measured the complexity of the eCFDs in terms of the $|\mathsf{Tp}|$, *i.e.,* the number of tuples in the pattern tableaux T_p, ranging from 10 to 500 pattern tuples. Note that each tuple itself is a constraint. The number of wildcards ('_'), positive domain constraints (S) and negative domain constraints ($\overline{S}$) in the pattern tuples are uniformly distributed.

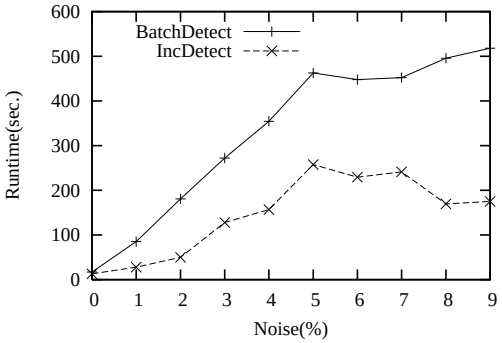
Our experiments were conducted on an Apple Xserve with 2.3GHz PowerPC dual CPU and 4GB of memory, and with a commercial DBMS installed. Each experiment was run five times and the average is reported here.

Experiment 1: Scalability. In the first set of experiments we evaluated the scalability of BATCHDETECT.

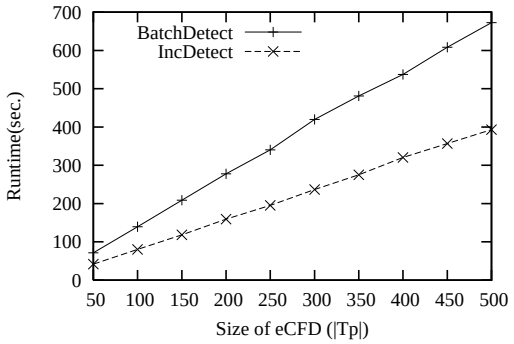
We first set $|\mathsf{Tp}| = 10$ and investigated the effect of varying $|\mathsf{D}|$ and noise% on the performance of BATCHDETECT. Fixing noise% = 5%, we varied $|\mathsf{D}|$ from 10k to 100k in 10k increments. Moreover, fixing $|\mathsf{D}| = 100$k, we varied noise% from 0% to 9% in 1% increments. The results are presented in Figs. 4.4(a) and 4.4(b). As expected, BATCHDETECT scales well *w.r.t.* the size of the datasets and the error rate.



(a) Scalability in |D|



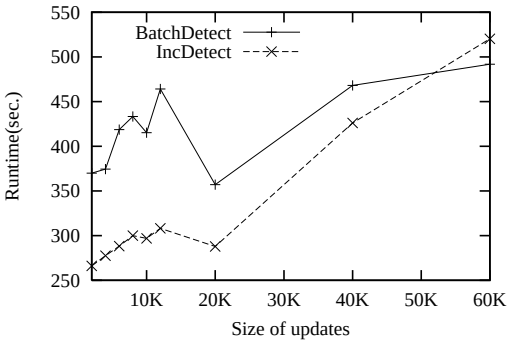
(b) Scalability in noise



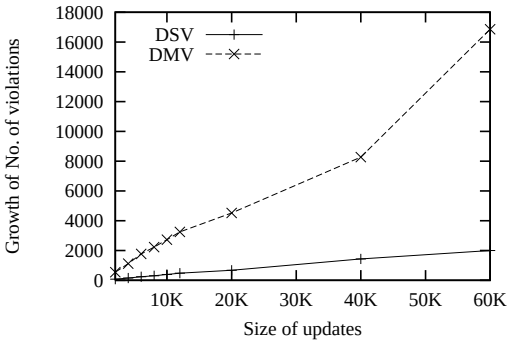
(c) Scalability in number of constraints

Figure 4.5: BATCHDETECT vs INCDETECT

We then set |D| = 100k and noise% = 5%, and studied the impact of varying the complexity of eCFDs of Σ on the cost of BATCHDETECT. We selected an eCFD from Σ and varied its |Tp| from 50 to 500 in 50 increments. As shown in Fig. 4.4(c) BATCHDETECT scales linearly in |Tp|.



(a) Scalability in size of updates



(b) Effect on number of violation changes

Figure 4.6: Effect of updates

Experiment 2: Incremental vs. Batch. In the second set of experiments we compared

the cost of incremental detection vs. its batch counterpart in response to database updates. We use $|\Delta D^+|$ and $|\Delta D^-|$ to indicate the number of tuples to be inserted into and deleted from D, respectively. We always ensure that ΔD^+ and ΔD^- do not overlap. As opposed to the first set of experiments, here BATCHDETECT was applied to the data after database updates are executed.

First, we fixed $|\Delta D^+|$ = 10k and $|\Delta D^-|$ = 10k and repeated the same set of experiment sets as above, *i.e.,* we investigated the scalability of INCDETECT in response to the size of datasets, the error rate and the complexity of eCFDs. The results are shown in Figs. 4.5(a), 4.5(b), and 4.5(c). In each figure we show the running time of INCDETECT and BATCHDETECT, in response to both tuple insertions and deletions. The results tell us that INCDETECT scales well *w.r.t.* $|D|$, noise% and $|Tp|$ and more importantly, performs better than BATCHDETECT. We note that the running time, reported in Figs. 4.5(a) and 4.5(b), not only depend on the size of the updates but also on which tuples are part of the update. This explains why the curves show some irregular behavior, although we averaged over different updates.

Second, we fixed $|D|$ = 100K, noise% = 5%, $|Tp|$ = 10 and varied $|\Delta D^+|$ and $|\Delta D^-|$ from 2k to 12k in 2k increments and from 20k to 60k in 20k increments. Note that $|D|$ is indeed fixed, since we delete and insert the same number of tuples. We compared the cost of INCDETECT vs. the cost of BATCHDETECT in response to the number of updates. As shown in Fig. 4.6(a), INCDETECT outperforms BATCHDETECT when the size of the updates is relatively small, and is slightly better for larger ones. However, as expected, BATCHDETECT outperforms INCDETECT for very large updates. Indeed, in our experiments this happens when around 50% of the data is updated. Overall, we may conclude that INCDETECT works extremely well and scales up in a similar way as BATCHDETECT.

Finally, in Fig. 4.6(b), we report the growth of the number of single (resp. multiple) tuple violations, denoted by DSV (resp. DMV), in the database before and after updates, for an increasing number of updates. On our datasets, we observed that the number of single-tuple violations grows linearly in the number of updates. However, the number of multiple-tuple violations increases dramatically for large updates. This also explains why BATCHDETECT performs better for large updates (see Fig. 4.6(a)). Indeed, maintaining the auxiliary information by INCDETECT incurs a large overhead when the number of violations changes too much.

Summary. We may conclude the following from our experimental evaluation. (1) BATCHDETECT and INCDETECT scale well *w.r.t.* when the dataset size, the error

rate and the complexity of eCFDs increase. (2) INCDETECT significantly outperforms BATCHDETECT in response to both tuple insertions and deletions, for reasonably-sized updates. (3) BATCHDETECT performs better than INCDETECT when more than 50% of the data is updated.

4.2 Detecting Inconsistencies with CFD^{c}s

In this section we develop SQL-based techniques for error detection based on CFD^{c}s. Consider a set Σ of CFD^{c}s defined over a relational schema R.

4.2.1 Encoding CFD^{c}s with Data Tables

We first show that, by extending the encoding of [FGJK08], all CFD^{c}s in Σ on a relational schema R can be encoded with *two data tables*, no matter how many dependencies are in the sets.

We encode all pattern tuples in the CFD^{c}s of Σ with two tables enc_L and enc_R, which encodes the patterns in LHS, and the patterns in RHS together with the cardinality constraints, respectively. More specifically, we associate a unique identifier cid with each CFD^c in Σ, and let enc_R consist of the following attributes: (a) cid, (b) each attribute A appearing in the LHS of some CFD^{c}s in Σ. Similarly, enc_R is defined, but with an extra attributes card storing the cardinality c of the CFD^c cid. Note that the arity of enc_L (resp. enc_R) is bounded by $|R|+1$ (resp. $|R|+2$), where $|R|$ is the arity of R.

We populate enc_L and enc_R as follows. For each CFD^c $\varphi = R(X \to Y,\ t_p, c)$ in Σ, we generate a distinct cid id_φ for it, and do the following.

- Add a tuple t_1 to enc_L such that (a) $t_1[\mathsf{cid}] = \mathsf{id}_\varphi$; (b) for each $A \in X$, $t_1[A] = t_p(A)$; and (c) $t_1[B]$ = 'null' for all other attributes B in enc_L.
- Similarly add a tuple t_2 to enc_R such that (a) $t_2[\mathsf{cid}] = \mathsf{id}_\varphi$; (b) for each $A \in Y$, $t_2[A] = t_p(A)$; $t_2[\mathsf{card}] = c$; and (d) $t_2[B]$ = 'null' for all other attributes B in enc_R.

Example 4.2.1: Consider the CFD^{c}s φ_1, φ_2 and φ_3 defined on the relational schema sale in Example 3.3.2 of Section 3.3.1 in Chapter 3, where

φ_1: $([\mathsf{country}, \mathsf{zip}] \to \mathsf{street},\ t_p^1,\ 1)$ with $t_p^1 = (\text{UK}, \_ \parallel \_)$,
φ_2: $(\mathsf{country} \to \mathsf{state},\ t_p^2,\ 1)$ with $t_p^2 = (\text{UK} \parallel \text{N/A})$, and
φ_3: $([\mathsf{FN}, \mathsf{LN}, \mathsf{street}, \mathsf{city}, \mathsf{state}, \mathsf{country}, \mathsf{zip}, \mathsf{type}] \to \mathsf{item},\ t_p^3,\ 2)$ with
$t_p^3 = (\_, \_, \_, \_, \_, \_, \_, \mathsf{sale} \parallel \_)$.

These three CFD^{c}s are encoded with tables shown in Fig. 4.7: (a) enc_L consists of attributes: cid, FN, LN, country, state, city, street, zip, and type; and (b) enc_R consists

(1) enc$_L$

cid	FN	LN	country	state	city	street	zip	type
1	null	null	UK	null	null	null	_	null
2	null	null	UK	null	null	null	null	null
3	_	_	_	_	_	_	_	sale

(2) enc$_R$

cid	state	street	item	card
1	null	_	null	1
2	N/A	null	null	1
3	null	null	_	2

Figure 4.7: Encoding example of CFD^{c}s

of attributes: cid, state, street, item, and card. One can easily reconstruct these CFD^{c}s from tables enc$_L$ and enc$_R$ by joining tuples with the same cid. □

4.2.2 SQL-based Detection Methods

Based on the synonym rules, *i.e.,* the relation R_c, we first compute a normalized relation as a preprocessing step. That is, (a) we partition all the constants appearing in R_c into equivalent classes such that $a \doteq b$ for two distinct elements in the same equivalent class, and $a \not\doteq b$ for two distinct elements in two distinct equivalent classes. (b) We choose a unique constant as a representation for each equivalent class. (c) After all data values are transformed into the representations of equivalent classes to which they belong if they appear in R_c, and are unchanged otherwise. Finally, we develop two SQL queries to check CFD^{c}s violations on the preprocessed data.

Below we first show how the two SQL queries $Q_{(i,sv)}$ and $Q_{(i,mv)}$ are generated for validating CFD^{c}s in Σ, which is an extension of the SQL techniques for CFDs discussed in [FGJK08].

(1) The query $Q_{(i,sv)}$ for detecting single-tuple violations of CFD^{c}s is given as follows, based on the data tables enc$_L$ and enc$_R$ that encode those CFD^{c}s in Σ.

The checking of single-tuple violations is exactly the same as the one of CFDs [FGJK08]. For the completeness of the story, we choose to present it here.

select $R.*$
from R, enc$_L$, enc$_R$
where enc$_L$.cid = enc$_R$.cid **and** $R.X \asymp$ enc$_L$ **and not** $R.Y \asymp$ enc$_R$

Here (a) $X = \{A_1, \ldots, A_g\}$ and $Y = \{B_1, \ldots, B_h\}$ are the sets of R attributes appearing in the LHS and the RHS of CFD^{c}s in Σ, respectively; (b) $R.X \asymp \mathsf{enc}_L$ is (**$\mathsf{enc}_L.A_1$ is null or $R.A_1 = \mathsf{enc}_L.A_1$**) **and** ... **and** (**$\mathsf{enc}_L.A_g$ is null or $R.A_g = \mathsf{enc}_L.A_g$**); and (c) **not** $R.Y \asymp \mathsf{enc}_R$ is (**$\mathsf{enc}_R.B_1$ is not null and $R.B_1 = \mathsf{enc}_R.B_1$**) **or** ... **or** (**$\mathsf{enc}_R.B_h$ is not null and $R.B_h = \mathsf{enc}_R.B_h$**).

Intuitively, $R.X \asymp \mathsf{enc}_L$ ensures that the R tuples selected match the LHS patterns of some CFD^{c}s in Σ; and $R.Y \asymp \mathsf{enc}_R$ checks the corresponding RHS patterns of these CFD^{c}s on R tuples. The query $Q_{(i,sv)}$ detects those R tuples, which satisfy the LHS of some CFD^{c}s, but not the RHS of those CFD^{c}s. That is, each of those tuples itself violates the CFD^{c}s in Σ .

(2) The query $Q_{(i,mv)}$ for detecting multiple-tuple violations of CFD^{c}s is given as follows, again based on the data tables enc_L and enc_R that encode CFD^{c}s in Σ.

select m.cid, m.card, $m.X$, **count(*)**
from macro m
group by m.cid, m.card, $m.X$
having count(*) $>$ m.card,

where macro stands for the following:

select distinct enc_R.cid, enc_R.card, **case**($R.X$), **case**($R.Y$)
from R, enc_L, enc_R
where enc_L.cid = enc_R.cid **and** $R.X \asymp \mathsf{enc}_L$

Here (a) for $X = \{A_1, \ldots, A_g\}$ and $Y = \{B_1, \ldots, B_h\}$ are the sets of attributes of R_i appearing in the LHS and RHS of CFD^{p}s in Σ, respectively;

(b) **case**($R.X$) is

(**case when** $\mathsf{enc}_L.A_1$ **is null then** '@' **else** $R.A_1$ **end**) **as** A_{1_L}, ...,
(**case when** $\mathsf{enc}_L.A_g$ **is null then** '@' **else** $R.A_g$ **end**) **as** A_{g_L};

(c) **case**($R.Y$) is

(**case when** $\mathsf{enc}_R.B_1$ **is null then** '@' **else** $R.B_1$ **end**) **as** B_{1_R}, ...,
(**case when** $\mathsf{enc}_R.B_h$ **is null then** '@' **else** $R.B_h$ **end**) **as** B_{h_R};

(d) $R.X \asymp \mathsf{enc}_L$ is the same as the one in the query $Q_{(i,sv)}$.

Intuitively, macro considers each CFD^c encoded in the tables enc_L and enc_R, and contains those R tuples that match the LHS of that CFD^p. For all attributes not appearing in the CFD^c, all the (possibly different) attribute values in the relation tuples are masked

and replaced by an '@' in macro. once we get the macro, the checking of multiple-tuple violations is straightforward, as indicated by the SQL query $Q_{(i,mv)}$.

Example 4.2.2: Using the coding of Fig. 4.7, two SQL query $Q_{(i,sv)}$ and $Q_{(i,mv)}$ for checking CFD^{c}s φ_1, φ_2 and φ_3 defined on the relational schema sale are given as follows:

$Q_{(i,sv)}$: **select** sale.$*$
from sale, enc$_L$, enc$_R$
where enc$_L$.cid = enc$_L$.cid **and** (enc$_L$.FN **is null or** sale.FN = enc$_L$.FN)
and (enc$_L$.LN **is null or** sale.LN = enc$_L$.LN)
and (enc$_L$.country **is null or** sale.country = enc$_L$.country)
and (enc$_L$.state **is null or** sale.state = enc$_L$.state)
and (enc$_L$.city **is null or** sale.city = enc$_L$.city)
and (enc$_L$.street **is null or** sale.street = enc$_L$.street)
and (enc$_L$.zip **is null or** sale.zip = enc$_L$.zip)
and (enc$_L$.type **is null or** sale.type = enc$_L$.type)
and ((enc$_R$.state **is not null and** sale.state <> enc$_R$.state)
or (enc$_R$.street **is not null and** sale.street <> enc$_R$.street)
or (enc$_R$.item **is not null and** sale.item <> enc$_R$.item))

Similarly, the query $Q_{(i,mv)}$ can be defined. □

4.3 Detecting Inconsistencies with CFD^{p}s and CIND^{p}s

If CFD^{p}s and CIND^{p}s are to be used as data quality rules, the first question we have to settle is how to effectively detect errors and inconsistencies as violations of these dependencies, by leveraging functionality supported by commercial DBMS. More specifically, consider a database schema $\mathcal{R} = (R_1, \ldots, R_n)$, where R_i is a relation schema for $i \in [1,n]$. The error detection problem is stated as follows.

The *error detection problem* is to find, given a set Σ of CFD^{p}s and CIND^{p}s defined on $\mathcal{R}$, and a database instance $D = (I_1, \ldots, I_n)$ of $\mathcal{R}$ as input, the subset $(I'_1, \ldots, I'_n)$ of D such that for each $i \in [1,n]$, $I'_i \subseteq I_i$ and each tuple in I'_i violates at least one CFD^p or CIND^p in Σ. We denote the set as vio(D,Σ), referred to it as *the violation set* of D *w.r.t.* Σ.

In this section we develop SQL-based techniques for error detection based on CFD^{p}s and CIND^{p}s. The main result of the section is as follows.

Theorem 4.3.1: *Given a set* Σ *of* CFD^p*s and* CIND^p*s defined on* $\mathcal{R}$ *and a database instance* D *of* $\mathcal{R}$*, where* $\mathcal{R} = (R_1, \ldots, R_n)$*, a set of* SQL *queries can be automatically generated such that (a) the collection of the answers to the* SQL *queries in* D *is* $\mathsf{vio}(D, \Sigma)$*, (b) the number and size of the set of* SQL *queries depend only on the number* n *of relations and their arities in* $\mathcal{R}$*, regardless of* Σ*.* □

We next present the main techniques for the query generation method. Let $\Sigma^i_{\mathsf{cfd^p}}$ be the set of all CFD^{p}s in Σ defined on the same relation schema R_i, and $\Sigma^{(i,j)}_{\mathsf{cind^p}}$ the set of all CIND^{p}s in Σ from R_i to R_j, for $i, j \in [1, n]$. We show the following. (a) The violation set $\mathsf{vio}(D, \Sigma^i_{\mathsf{cfd^p}})$ can be computed by *two* SQL queries. (b) Similarly, $\mathsf{vio}(D, \Sigma^{(i,j)}_{\mathsf{cind^p}})$ can be computed by a *single* SQL query. (c) These SQL queries encode pattern tableaux of CFD^{p}s (CIND^{p}s) with data tables, and hence their sizes are independent of Σ. From these Theorem 4.3.1 follows immediately.

4.3.1 Encoding CFD^{p}s and CIND^{p}s with Data Tables

We first show the following, by extending the encoding of [FGJK08, BFGM08]. (a) The pattern tableaux of all CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$ can be encoded with *three data tables*, and (b) the pattern tableaux of all CIND^{p}s in $\Sigma^{(i,j)}_{\mathsf{cind^p}}$ can be represented as *four data tables*, no matter how many dependencies are in the sets and how large they are.

Encoding CFD^{p}s. We encode all pattern tableaux in $\Sigma^i_{\mathsf{cfd^p}}$ with three tables enc_L, enc_R and $\mathsf{enc}_{\neq}$, where enc_L (resp. enc_R) encodes the non-negation ($=, <, \leq, >, \geq$) patterns in LHS (resp. RHS), and $\mathsf{enc}_{\neq}$ encodes those negation ($\neq$) patterns. More specifically, we associate a unique identifier cid with each CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$, and let enc_L consist of the following attributes: (a) cid, (b) each attribute A appearing in the LHS of some CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$, (c) its six companion attributes $A_>$, $A_\geq$, $A_<$, $A_\leq$, and $\mathsf{A_b}$. That is, for each attribute, there are six columns in enc_L including one for each non-negation operator, and one indicating that the attribute A appears in the LHS of the CFD^p with identifier cid when $\mathsf{A_b}$ = '1', or not otherwise. Similarly, enc_R is defined. We use an $\mathsf{enc}_{\neq}$ tuple to encode a pattern $A \neq c$ in a CFD^p, consisting of cid, att, pos, and val, encoding the CFD^p id, the attribute A, the position ('LHS' or 'RHS'), and the constant c, respectively. Note that the arity of enc_L (enc_R) is bounded by $6 * |R_i| + 1$, where $|R_i|$ is the arity of R_i, and the arity of $\mathsf{enc}_{\neq}$ is 4.

Before we populate these tables, let us first describe a preferred form of CFD^{p}s that would simplify the analysis to be given. Consider a CFD^p $\varphi = R(X \rightarrow Y, T_p)$. If φ is not satisfiable we can simply drop it from Σ. Otherwise it is equivalent to a CFD^p

(1) enc_L

cid	sale	price	sale_b	$\mathsf{price}_\mathsf{b}$	$\mathsf{price}_>$	$\mathsf{price}_\leq$
2	T	null	1	null	null	null
3	F	_	1	1	20	40
4	T	null	1	null	null	null

(2) enc_R

cid	shipping	price	$\mathsf{shipping}_\mathsf{b}$	$\mathsf{price}_\mathsf{b}$	$\mathsf{price}_\geq$	$\mathsf{price}_<$
2	0	null	1	null	null	null
3	6	null	1	null	null	null
4	null	_	null	1	2.99	9.99

(3) $\mathsf{enc}_\neq$

cid	pos	att	val

Figure 4.8: Encoding example of CFD^{p}s

$\varphi' = R(X \to Y,\ T'_p)$ such that for any pattern tuples t_p, t'_p in T'_p and for any attribute A in $X \cup Y$, (a) if $t_p[A]$ is op a and $t'_p[A]$ is op b, where op is not $\neq$, then $a = b$, (b) if $t_p[A]$ is '_' then so is $t'_p[A]$. That is, for each non-negation op (resp. _), there is a *unique* constant a such that $t_p[A]$ = 'op a' (resp. $t_p[A] =$ _) is the only op (resp. _) pattern appearing in the A column of T'_p. We refer to $t_p[A]$ as $T'_p(\mathsf{op}, A)$ (resp. $T'_p(\_, A)$), and consider *w.l.o.g.* CFD^{p}s of this form only. Note that there are possibly multiple $t_p[A] \neq c$ patterns in T'_p.

We populate enc_L, enc_R and $\mathsf{enc}_\neq$ as follows. For each CFD^p $\varphi = R(X \to Y,\ T_p)$ in $\Sigma^i_{\mathsf{cfd^p}}$, we generate a distinct cid id_φ for it, and do the following.

- Add a tuple t_1 to enc_L such that (a) $t_1[\mathsf{cid}] = \mathsf{id}_\varphi$; (b) for each $A \in X$, $t_1[A]$ = _ if $T'_p(\_, A)$ is '_', for each non-negation predicate op, $t_1[A_{\mathsf{op}}]$ = 'a' if $T'_p(\mathsf{op}, A)$ is 'op a', and $t_1[A_b]$ = '1'; (c) we let $t_1[B]$ = 'null' for all other attributes B in enc_L.
- Similarly add a tuple t_2 to enc_R for attributes in Y.
- For each attribute $A \in X \cup Y$ and each $\neq a$ pattern in $T_p[A]$, add a tuple t to $\mathsf{enc}_\neq$ such that $t[\mathsf{cid}] = \mathsf{id}_\varphi$, $t[\mathsf{att}]$ = 'A', $t[\mathsf{val}]$ = 'a', and $t[\mathsf{pos}]$ = 'LHS' (resp. $t[\mathsf{pos}]$ = 'RHS') if attribute A appears in X (resp. Y).

Example 4.3.1: Consider the CFD^{p}s φ_2, φ_3 and φ_4 defined on the relational schema item in Example 3.4.2 of Section 3.4.2 in Chapter 3, where

φ_2: (sale $\to$ shipping, $\{t_p^2\}$), where t_p^2 is (= T $\|$ = 0);
φ_3: ([sale, price] $\to$ shipping, $\{t_p^{3,1}, t_p^{3,2}\}$), where

(1) enc

cid	state_L	state_R
1	1	1
2	1	1

(2) enc_L

cid	type	state
1	_	null
2	_	DL

(3) enc_R

cid	rate
1	null
2	0

(4) $\text{enc}_{\neq}$

cid	pos	att	val
1	LHS	type	art
2	LHS	type	art

Figure 4.9: Encoding example of CIND^{p}s

$t_p^{3,1}$ is (= F, $> 20 \parallel = 6$) and $t_p^{3,2}$ is (= F, $\leq 40 \parallel = 6$); and
φ_4: (sale $\rightarrow$ price, $\{t_p^{4,1}, t_p^{4,2}\}$), where $t_p^{4,1}$ is (= F $\parallel \geq 2.99$) and $t_p^{4,2}$ is (= F $\parallel < 9.99$).

These CFD^{p}s are encoded with tables shown in Fig. 4.8: (a) enc_L consists of attributes: cid, sale, price, sale_b, price_b, $\text{price}_>$ and $\text{price}_\leq$; (b) enc_R consists of cid, shipping, price, shipping_b, price_b, $\text{price}_\geq$ and $\text{price}_<$; those attributes in a table with only 'null' pattern values do not contribute to error detection, and are thus omitted; (c) $\text{enc}_{\neq}$ is empty since all these CFD^{p}s have no negation patterns. One can easily reconstruct these CFD^{p}s from tables enc_L, enc_R and $\text{enc}_{\neq}$ by collating tuples based on cid. □

Encoding CIND^{p}s. All CIND^{p}s in $\Sigma_{\text{cind}^\text{p}}^{(i,j)}$ can be encoded with four tables enc, enc_L, enc_R and $\text{enc}_{\neq}$. Here enc_L (resp. enc_R) and $\text{enc}_{\neq}$ encode non-negation patterns on relation R_i (resp. R_j) and negation patterns on relations R_i or R_j, respectively, along the same lines as their counterparts for CFD^{p}s. We use enc to encode the INDs *embedded* in CIND^{p}s, which consists of the following attributes: (1) cid representing the identifier of a CIND^p, and (2) those X attributes of R_i and Y attributes of R_j appearing in some CIND^{p}s in $\Sigma_{\text{cind}^\text{p}}^{(i,j)}$. Note that the number of attributes in enc is bounded by $|R_i|+|R_j|+1$, where $|R_i|$ is the arity of R_i.

For each CIND^p $\psi = (R_i[A_1 \ldots A_m;\ X_p] \subseteq R_j[B_1 \ldots B_m;\ Y_p],\ T_p)$ in $\Sigma_{\text{cind}^\text{p}}^{(i,j)}$, we generate a distinct cid id_ψ for it, and do the following.

- Add tuples t_1 and t_2 to enc_L and enc_R based on attributes X_p and Y_p, respectively, along the same lines as their CFD^p counterpart.
- Add tuples to $\text{enc}_{\neq}$ in the same way as their CFD^p counterparts.
- Add tuple t to enc such that $t[\text{cid}] = \text{id}_\psi$. For each $k \in [1,m]$, let $t[A_k] = t[B_k] = k$, and $t[A]$ = 'null' for the rest attributes A of enc.

Example 4.3.2: Consider the CIND^{p}s ψ_1 and ψ_2 defined on the relational schemas item and tax in Example 3.4.4 of Section 3.4.3 in Chapter 3, where

ψ_1: (item [state; type] $\subseteq$ tax [state; nil], $\{t_p^1\}$), where t_p^1 is ($\neq$ art $\|$ nil); and
ψ_2: (item [state; type, state] $\subseteq$ tax [state; rate], $\{t_p^2\}$), where t_p^2 is ($\neq$ art, = DL $\| = 0$)

Figure 4.8 shows the coding of CIND^{p}s ψ_1 and ψ_2. We use $\mathsf{state_L}$ and $\mathsf{state_R}$ in enc to denote the occurrences of attribute state in item and tax, respectively. In tables enc_L and enc_R, attributes with only 'null' patterns are omitted, for the same reason as for CFD^{p}s mentioned above. □

Putting these together, it is easy to verify that at most $O(n^2)$ data tables are needed to encode dependencies in Σ, regardless of the size of Σ. Recall that n is the number of relations in database $\mathcal{R}$.

4.3.2 SQL-based Detection Methods

We next show how to generate SQL queries based on the encoding above. For each $i \in [1,n]$, we generate *two* SQL queries that, when evaluated on the I_i table of D, find $\mathsf{vio}(D, \Sigma^i_{\mathsf{cfd^p}})$. Similarly, for each $i, j \in [1,n]$, we generate a *single* SQL query $Q_{(i,j)}$ that, when evaluated on (I_i, I_j) of D, returns $\mathsf{vio}(D, \Sigma^{(i,j)}_{\mathsf{cind^p}})$. Putting these query answers together, we get $\mathsf{vio}(D, \Sigma)$, the violation set of D *w.r.t.* Σ.

Detecting CFD^p violations. Below we first show how the two SQL queries $Q_{(i,sv)}$ and $Q_{(i,mv)}$ are generated for validating CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$, which is an extension of the SQL techniques for CFDs discussed in [FGJK08, BFGM08].

(1) The query $Q_{(i,sv)}$ for detecting single-tuple violations of $\Sigma^i_{\mathsf{cfd^p}}$ is given as follows, based on the data tables enc_L, enc_R and $\mathsf{enc}_{\neq}$ that encode CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$.

select $R_i.*$
from R_i, enc_L L, enc_R R, $\mathsf{enc}_{\neq}$ N
where $L.\mathsf{cid} = R.\mathsf{cid}$ **and** ($R_i.X \asymp L$ **and** $R_i.X \asymp N$) **and not** ($R_i.Y \asymp R$ **and** $R_i.Y \asymp N$)

Here (a) $X = \{\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{g}\}$ and $Y = \{\mathsf{B}_1, \ldots, \mathsf{B}_\mathsf{h}\}$ are the sets of attributes of R_i appearing in the LHS and RHS of CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$, respectively;

(b) $R_i.X \asymp L$ is the conjunction of

$L.A_k$ **is null or** $R_i.A_k = L.A_k$ **or** ($L.A_k$ = '_' **and**
($L.A_{k_>}$ **is null or** $R_i.A_k > L.A_{k_>}$) **and** ($L.A_{k_\geq}$ **is null or** $R_i.A_k \geq L.A_{k_\geq}$) **and**
($L.A_{k_<}$ **is null or** $R_i.A_k < L.A_{k_<}$) **and** ($L.A_{i_\leq}$ **is null or** $R_i.A_k \leq L.A_{i_\leq}$))

for $k \in [1,g]$;

(c) $R_j.Y \asymp R$ is defined similarly for attributes in Y;

(d) $R_i.X \asymp N$ is a shorthand for the conjunction below, for $k \in [1,g]$:

not exists (**select** $*$
from N
where L.cid = N.cid **and** N.pos = 'LHS' **and**
N.att = 'A_k' **and** $R_i.A_k = N$.val);

(e) $R_i.Y \asymp N$ is defined similarly, but with N.pos = 'RHS'.

Intuitively, $R_i.X \asymp L$ **and** $R_i.X \asymp N$ ensure that the R_i tuples selected match the LHS patterns of some CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$; and $R_i.Y \asymp R$ **and** $R_i.Y \asymp N$ check the corresponding RHS patterns of these CFD^{p}s on R_i tuples. The query $Q_{(i,sv)}$ detects those R_i tuples, which satisfy the LHS of some CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$, but do not satisfy the RHS of those CFD^{p}s. That is, each of those tuples itself violates the CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$.

(2) The query $Q_{(i,mv)}$ for detecting multiple-tuple violations of $\Sigma^i_{\mathsf{cfd^p}}$ is given as follows, again based on the data tables enc, enc_L, enc_R and $\mathsf{enc}_{\neq}$ that encode CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$.

select m.cid, $m.X$, **count(*)**
from macro m
group by m.cid, $m.X$
having count(*) > 1,

where macro stands for the following:

select distinct L.cid, **case**($R_i.X$), **case**($R_i.Y$)
from R_i, $\mathsf{enc}_L\ L$, $\mathsf{enc}_R\ R$, $\mathsf{enc}_{\neq}\ N$
where L.cid = R.cid **and** $R_i.X \asymp L$ **and** $R_i.X \asymp N$

Here (a) for $X = \{\mathsf{A}_1, \ldots, \mathsf{A}_\mathsf{g}\}$ and $Y = \{\mathsf{B}_1, \ldots, \mathsf{B}_\mathsf{h}\}$ are the sets of attributes of R_i appearing in the LHS and RHS of CFD^{p}s in $\Sigma^i_{\mathsf{cfd^p}}$, respectively;

(b) **case**($R_i.X$) is

(**case** $\mathsf{A}_{1_\mathsf{b}}$ **when** 1 **then** $R_i.A_1$ **else** '@' **end**) **as** A_{1_L}, ...,
(**case** $\mathsf{A}_{\mathsf{g}_\mathsf{b}}$ **when** 1 **then** $R_i.A_g$ **else** '@' **end**) **as** A_{g_L};

(c) **case**($R_i.Y$) is

(**case** $\mathsf{B}_{1_\mathsf{b}}$ **when** 1 **then** $R_i.B_1$ **else** '@' **end**) **as** B_{1_R}, ...,
(**case** $\mathsf{B}_{\mathsf{h}_\mathsf{b}}$ **when** 1 **then** $R_i.B_h$ **else** '@' **end**) **as** B_{h_R};

(d) $R_i.X \asymp L$ and $R_i.X \asymp N$ are the same as the ones in the query $Q_{(i,sv)}$.

Intuitively, macro considers each CFD^p encoded in the tables enc_L, enc_R R, and $\mathsf{enc}_{\neq}$, and contains those R_i tuples that match the LHS of that CFD^p. For all attributes not appearing in the CFD^p, all the (possibly different) attribute values in the relation tuples are masked and replaced by an '@' in macro. once we get the macro, the

checking of multiple-tuple violations is straightforward, as indicated by the SQL query $Q_{(i,mv)}$.

Example 4.3.3: Using the coding of Fig. 4.8, two SQL query $Q_{(i,sv)}$ and $Q_{(i,mv)}$ for checking CFD^{p}s φ_2, φ_2 and φ_4 defined on the relational schema item in Fig. 3.5 are given as follows:

$Q_{(i,sv)}$: **select** item.$*$
from item, $\mathsf{enc}_L\ L$, $\mathsf{enc}_R\ R$, $\mathsf{enc}_{\neq}\ N$
where L.cid = R.cid **and** (L.sale **is null or** item.sale = L.sale **or** L.sale = '_')
and not exists (**select ***
from N
where N.cid = L.cid **and** N.pos = 'LHS' **and** N.att = 'sale')
and (L.price **is null or** item.price = L.price **or** (L.price = '_' **and** ($L.\mathsf{price}_{>}$ **is null**
or item.price $>$ L.price) **and** ($L.\mathsf{price}_{\leq}$ **is null or** item.price $\leq$ L.price))
and not exists (**select ***
from N
where N.cid = L.cid **and** N.pos = 'LHS' **and** N.att = 'price')
and (**not** (R.shipping **is null or** item.shipping = L.shipping **or** L.shipping = '_')
or exists (**select ***
from N
where N.cid = L.cid **and** N.pos = 'RHS' **and** N.att = 'shipping')
or not (R.price **is null or** item.price = L.price **or** L.price = '_')
or exists (**select ***
from N
where N.cid = L.cid **and** N.pos = 'RHS' **and** N.att = 'price'))

Similarly, the query $Q_{(i,mv)}$ can be defined.

The SQL queries generated for error detection can be simplified as follows. As shown in Example 4.3.4, when checking patterns imposed by enc, enc_L or enc_R, the queries need not consider attributes A if $t[A]$ is 'null' for each tuple t in the table. Similarly, if an attribute A does not appear in any tuple in $\mathsf{enc}_{\neq}$, the queries need not check A either. From this, it follows that we do not even need to generate those attributes with only 'null' patterns for data tables enc, enc_L or enc_R when encoding CFD^{p}s or $Q_{(i,mv)}$. □

Detecting CIND^p violations. Then, we show how the SQL query $Q_{(i,j)}$ is generated for validating CIND^{p}s in $\Sigma^{(i,j)}_{\mathsf{cind^p}}$), which has not been studied by previous work.

The query $Q_{(i,j)}$ for the validation of $\Sigma^{(i,j)}_{\mathsf{cind^p}}$ is given as follows, which capitalizes on the data tables enc, enc_L, enc_R and $\mathsf{enc}_{\neq}$ that encode CIND^{p}s in $\Sigma^{(i,j)}_{\mathsf{cind^p}}$.

select $R_i.*$
from R_i, enc_L L, $\mathsf{enc}_{\neq}$ N
where $R_i.X \asymp L$ **and** $R_i.X \asymp N$ **and not exists** (
  select $R_j.*$
  from R_j, enc H, enc_R R, $\mathsf{enc}_{\neq}$ N
  where $R_i.X = R_j.Y$ **and** $L.\mathsf{cid} = R.\mathsf{cid}$ **and** $L.\mathsf{cid} = H.\mathsf{cid}$ **and** $R_j.Y \asymp R$ **and** $R_j.Y \asymp N$)

Here (a) $X = \{\mathsf{A}_1, \ldots, \mathsf{A}_{m1}\}$ and $Y = \{\mathsf{B}_1, \ldots, \mathsf{B}_{m2}\}$ are the sets of attributes of R_i and R_j appearing in $\Sigma^{(i,j)}_{\mathsf{cind^p}}$, respectively; (b) $R_i.X \asymp L$ is the conjunction of

$L.A_k$ **is null or** $R_i.A_k = L.A_k$ **or** ($L.A_k$ = '_' **and**
  ($L.A_{k_>}$ **is null or** $R_i.A_k > L.A_{k_>}$) **and** ($L.A_{k_\geq}$ **is null or** $R_i.A_k \geq L.A_{k_\geq}$) **and**
  ($L.A_{k_<}$ **is null or** $R_i.A_k < L.A_{k_<}$) **and** ($L.A_{i_\leq}$ **is null or** $R_i.A_k \leq L.A_{i_\leq}$))

for $k \in [1, m_1]$; (c) $R_j.Y \asymp R$ is defined similarly for attributes in Y; (d) $R_i.X \asymp N$ is a shorthand for the conjunction below, for $k \in [1, m_1]$:

not exists (**select** $*$
  from N
  where $L.\mathsf{cid} = N.\mathsf{cid}$ **and** $N.\mathsf{pos}$ = 'LHS' **and** $N.\mathsf{att}$ = 'A_k' **and** $R_i.A_k = N.\mathsf{val}$);

(e) $R_j.Y \asymp N$ is defined similarly, but with $N.\mathsf{pos}$ = 'RHS' ; (f) $R_i.X = R_j.Y$ represents the following: for each A_k ($k \in [1, m_1]$) and each B_l ($l \in [1, m_2]$), ($H.A_k$ **is null or** $H.B_l$ **is null or** $H.B_l \neq H.A_k$ **or** $R_i.A_k = R_j.B_l$).

Intuitively, (a) $R_i.X \asymp L$ **and** $R_i.X \asymp N$ ensure that the R_i tuples selected match the LHS patterns of some CIND^{p}s in $\Sigma^{(i,j)}_{\mathsf{cind^p}}$; (b) $R_j.Y \asymp R$ **and** $R_j.Y \asymp N$ check the corresponding RHS patterns of these CIND^{p}s on R_j tuples; (c) $R_i.X = R_j.Y$ enforces the *embedded* INDs; (d) $L.\mathsf{cid} = R.\mathsf{cid}$ **and** $L.\mathsf{cid} = H.\mathsf{cid}$ assure that the LHS and RHS patterns in the same CIND^p are correctly collated; and (e) **not exists** in Q ensures that the R_i tuples selected violate CIND^{p}s in $\Sigma^{(i,j)}_{\mathsf{cind^p}}$.

Example 4.3.4: Using the coding of Fig. 4.9, an SQL query Q for checking CIND^{p}s ψ_1 and ψ_2 of Fig. 3.6 is given as follows:

select $R_1.*$
from item R_1, enc_L L, $\mathsf{enc}_{\neq}$ N
where ($L.\mathsf{type}$ **is null or** $R_1.\mathsf{type} = L.\mathsf{type}$ **or** $L.\mathsf{type}$ = '_')
  and not exists (**select** *
    from N
    where $N.\mathsf{cid} = L.\mathsf{cid}$ **and** $N.\mathsf{pos}$ = 'LHS' **and** $N.\mathsf{att}$ = 'type')
  and ($L.\mathsf{state}$ **is null or** $R_1.\mathsf{state} = L.\mathsf{state}$ **or** $L.\mathsf{state}$ = '_')

and not exists (**select** *
from N
where N.cid = L.cid **and** N.pos = 'LHS' **and** N.att = 'state'
and R_1.state =N.val)
and not exists (**select** R_2.*
from tax R_2, enc H, enc$_R$ R
where (H.state$_\mathsf{L}$ **is null or** H.state$_\mathsf{R}$ **is null or** H.state$_\mathsf{L}$! = H.state$_\mathsf{R}$
or R_2.state = R_1.state) **and** L.cid = H.cid **and** L.cid = R.cid
and (R.rate **is null or** R_2.rate = R.rate **or** R.rate = '_')
and not exists (**select** *
from N
where N.cid = R.cid **and** N.pos = 'RHS'
and N.att = 'rate' **and** R_2.rate =N.val))

The SQL queries generated for error detection can be simplified as follows. As shown in Example 4.3.4, when checking patterns imposed by enc, enc$_L$ or enc$_R$, the queries need not consider attributes A if $t[A]$ is 'null' for each tuple t in the table. Similarly, if an attribute A does not appear in any tuple in enc$_{\neq}$, the queries need not check A either. From this, it follows that we do not even need to generate those attributes with only 'null' patterns for data tables enc, enc$_L$ or enc$_R$ when encoding CIND^{p}s or CFD^{p}s. □

Remark. As pointed out earlier in Section 3.4, CIND^{p}s subsume CINDs. Therefore, the SQL-techniques for CIND^{p}s can be directly applied to CIND^{p}s. Indeed, the SQL queries for CINDs is much simpler by removing irrelevant sub-queries from the above SQL queries.

Chapter 5

Propagating Functional Dependencies with Conditions

The dependency propagation problem is to determine, given a view defined on data sources and a set of dependencies on the sources, whether another dependency is guaranteed to hold on the view. This chapter investigates dependency propagation for recently proposed conditional functional dependencies (CFDs). The need for this study is evident in data integration, exchange and cleaning since dependencies on data sources often only hold *conditionally* on the view. We investigate dependency propagation for views defined in various fragments of relational algebra, CFDs as view dependencies, and for source dependencies given as either CFDs or traditional functional dependencies (FDs). (a) We establish lower and upper bounds, *all matching*, ranging from PTIME to undecidable. These not only provide the *first* results for CFD propagation, but also extend the classical work of FD propagation by giving new complexity bounds in the presence of finite domains. (b) We provide the first algorithm for computing a minimal cover of *all* CFDs propagated via SPC views; the algorithm has the same complexity as one of the most efficient algorithms for computing a cover of FDs propagated via a projection view, despite the increased expressive power of CFDs and SPC views. (c) We experimentally verify that the algorithm is efficient.

5.1 Introduction

The prevalent use of the Web has made it possible to exchange and integrate data on an unprecedented scale. A natural question in connection with data exchange and integration concerns whether dependencies that hold on data sources still hold on the

(a) Instance D_1 of R_1, for UK customers

	AC	phn	name	street	city	zip
t_1:	20	1234567	Mike	Portland	LDN	W1B 1JL
t_2:	20	3456789	Rick	Portland	LDN	W1B 1JL

(b) Instance D_2 of R_2, for US customers

	AC	phn	name	street	city	zip
t_3:	610	3456789	Joe	Copley	Darby	19082
t_4:	610	1234567	Mary	Walnut	Darby	19082

(c) Instance D_3 of R_3, for customers in Netherlands

	AC	phn	name	street	city	zip
t_5:	20	3456789	Marx	Kruise	Amsterdam	1096
t_6:	36	1234567	Bart	Grote	Almere	1316

Figure 5.1: Instances of R_1, R_2, R_3 relations

target data (*i.e.,* data transformed via mapping from the sources). As dependencies (*a.k.a.* integrity constraints) specify a fundamental part of the semantics of the data, one wants to know whether or not the dependencies are propagated from the sources via the mapping, *i.e.,* whether the mapping preserves information.

This is one of the classical problems in database research, referred to as the *dependency propagation problem*. It is to determine, given a view (mapping) defined on data sources and dependencies that hold on the sources, whether or not another dependency is guaranteed to hold on the view? We refer to the dependencies defined on the sources as *source dependencies*, and those on the view as *view dependencies*.

This problem has been extensively studied when source and view dependencies are functional dependencies (FDs), for views defined in relational algebra (*e.g.,* [Fag82, FJT83, Klu80, KP82, Got87]). It is considered an issue already settled in the 1980s.

It turns out that while many source FDs may not hold on the view as they are, they do hold on the view under *conditions*. That is, source FDs are indeed propagated to the view, not as standard FDs but as FDs with conditions. The FDs with conditions are in the form of *conditional functional dependencies* (CFDs) recently proposed [FGJK08], as shown below.

Example 5.1.1: Consider three data sources R_1, R_2 and R_3, containing information

about customers in the UK, US and Netherlands, respectively. To simplify the presentation we assume that these data sources have a uniform schema:

R_i(AC: string, phn: string, name: string, street: string, city: string, zip: string)

Each tuple in an R_i relation specifies a customer's information (area code AC, phone phn, name and address (street, city, zip code)), for $i \in [1,3]$. Example instances D_1, D_2 and D_3 of R_1, R_2 and R_3 are shown in Fig. 5.1.

Consider the following FDs defined on the UK and Holland sources: in instances of R_1, zip code uniquely determines street (f_1), and area code uniquely determines city (f_2); moreover, area code determines city in R_3 data (f_3).

f_1: $R_1(\mathsf{zip} \rightarrow \mathsf{street})$, f_2: $R_1(\mathsf{AC} \rightarrow \mathsf{city})$, f_3: $R_3(\mathsf{AC} \rightarrow \mathsf{city})$.

Define a view V with query $Q_1 \cup Q_2 \cup Q_3$ to integrate the data from the three sources, where Q_1 is

select AC, phn, name, street, city, zip, '44' as CC **from** R_1

Define Q_2 and Q_3 by substituting '01' and '31' for '44', R_2 and R_3 for R_1 in Q_1, respectively. The target schema R has all the attributes in the sources and a country-code attribute CC (44, 01, 31 for the UK, US and Netherlands, respectively).

Now one wants to know whether f_1 on the R_1 source still holds on the target data (view). The answer is negative: Figure 5.1 tells us that the view violates f_1 due to tuples t_3, t_4 extracted from D_2; indeed, in the US, zip does not determine street. That is, f_1 is *not* propagated to the view as an FD. In contrast, the following CFD [FGJK08] holds on the view:

φ_1: $R([\mathsf{CC} = \text{'44'}, \mathsf{zip}] \rightarrow [\mathsf{street}])$.

That is, for UK customers in the view, zip code uniquely determines street. In other words, φ_1 is an "FD" with a condition: it is to hold only on the subset of tuples in the view that satisfies the pattern CC = '44', rather than on the entire view. It cannot be expressed as a standard FD.

Similarly, from f_2 and f_3 one *cannot* derive a standard FD on the view to assert that "area code uniquely determines city". Indeed, from tuples t_1 and t_5 in Fig. 5.1 we can see that 20 is an area code in both the UK and Holland, for London and Amsterdam, respectively. However, not all is lost: the following CFDs are propagated from f_2 and f_3 via the view:

φ_2: $R([\mathsf{CC} = \text{'44'}, \mathsf{AC}] \rightarrow [\mathsf{city}])$,
φ_3: $R([\mathsf{CC} = \text{'31'}, \mathsf{AC}] \rightarrow [\mathsf{city}])$.

That is, f_2 and f_3 hold conditionally on the view: area code determines city for tuples with CC = '44' (φ_2) or CC = '31' (φ_3). In other words, the semantics specified by the FDs on the sources is preserved by the CFDs on the view.

Furthermore, given the following CFDs on the sources:

cfd_1: $R_1([\mathsf{AC} = \text{'20'}] \rightarrow [\mathsf{city} = \text{'LDN'}])$,
cfd_2: $R_3([\mathsf{AC} = \text{'20'}] \rightarrow [\mathsf{city} = \text{'Amsterdam'}])$,

then the following CFDs are propagated to the view:

φ_4: $R([\mathsf{CC} = \text{'44'}, \mathsf{AC} = \text{'20'}] \rightarrow [\mathsf{city} = \text{'LDN'}])$,
φ_5: $R([\mathsf{CC} = \text{'31'}, \mathsf{AC} = \text{'20'}] \rightarrow [\mathsf{city} = \text{'Amsterdam'}])$,

which carry patterns of semantically related constants. □

No previous algorithms developed for FD propagation are capable of deriving these CFDs from the given source FDs via the view. This highlights the need for investigating dependency propagation, for CFDs as view dependencies.

Applications. The study of dependency propagation is not only of theoretical interest, but also important in practice.

(1) Data exchange [Kol05a]. Recall Example 5.1.1. Suppose that the target schema R and CFDs φ_2 and φ_3 are predefined. Then the propagation analysis assures that the view definition V is a schema mapping from (R_1, R_2, R_3) to R, *i.e.,* for any source instances D_1 and D_3 of R_1 and R_3 that satisfy the FDs f_2 and f_3, respectively, and for any source instance D_2 of R_2, the view $V(D_1, D_2, D_3)$ is an instance of the target schema R and is guaranteed to satisfy φ_2 and φ_3.

(2) Data integration [Len02]. Suppose that V is a mapping in an integration system, which defines a global view of the sources. Then certain view updates, *e.g.,* insertion of a tuple t with CC = '44', AC = '20' and city = 'EDI', can be rejected without checking the data, since it violates the CFD φ_4 propagated from the sources.

(3) Data cleaning. In contrast to FDs that were developed for schema design, CFDs were proposed for data cleaning [FGJK08]. Suppose that CFDs φ_1–φ_5 are defined on the target database, for checking the consistency of the data. Then propagation analysis assures that one need not validate these CFDs against the view V. In contrast, if in addition, an FD φ_6: $R(\mathsf{CC}, \mathsf{AC}, \mathsf{phn} \rightarrow \mathsf{street}, \mathsf{city}, \mathsf{zip})$ is also defined on the target, then φ_6 has to be validated against the view since it is not propagated from the source dependencies.

Contributions. In response to the practical need, we provide the *first results* for dependency propagation when view dependencies are CFDs. We study views expressed

in various fragments of relational algebra (RA), and source dependencies expressed either as traditional FDs or CFDs.

(1) *Complexity bounds*. We provide a complete picture of complexity bounds on dependency propagation, for source FDs and source CFDs, and for various fragments of RA views. Furthermore, we study the problem in two settings: (a) *the infinite-domain setting*: in the absence of finite-domain attributes in a schema, and (b) *the general setting* where finite-domain attributes may be present. We establish upper and lower bounds, *all matching*, for all these cases, ranging from polynomial time (PTIME) to undecidable. We show that in many cases CFD propagation retains the same complexity as its FD counterpart, but in some cases CFDs do make our lives harder by incurring extra complexity.

Previous work on dependency propagation assumes the infinite-domain setting. It is known that FD propagation is in PTIME for SPCU views involving infinite-domain attributes only [AHV95] (union of conjunctive queries, defined with selection, projection, Cartesian product and union operators). In real world, however, it is common to find attributes with a finite domain, *e.g.,* Boolean, date, etc. It is hence necessary to study the dependency propagation problem *in the presence of finite-domain attributes*, and get the complexity right in the general setting.

In light of this we study the analysis of dependency propagation in the general setting. We show that the presence of finite-domain attributes complicates the analysis, even for *source* FDs and *view* FDs. Indeed, while FD propagation is in PTIME for SPCU views in the infinite-domain setting, this is *no longer* the case in the general setting: the problem already becomes coNP-complete for SC views, source FDs and view FDs! This intractability is unfortunately what one often has to cope with in practice.

To our knowledge this work is the first effort to study the dependency propagation problem in the general setting.

(2) *Algorithms for computing a propagation cover.* In many applications one wants not only to know whether a given view dependency is propagated from source dependencies, but also to find a *cover* of *all* view dependencies propagated. From the cover all view dependencies can be deduced via implication analysis. This is needed for, *e.g.,* processing view updates and detecting inconsistencies, as shown by the data integration and data cleaning examples given above.

Although important, this problem is rather difficult. It is known [FJT83] that even for certain FDs and views defined with a single projection operator, a minimal cover of all view FDs propagated is sometimes necessarily exponentially large, in the infinite-

domain setting. A typical method to find a cover is by first computing the closure of all source FDs, and then projecting the closure onto the view schema. While this method always takes exponential time, it is the algorithm recommended by database textbooks [SK86, Ull82].

Already hard for FDs and projection views, the propagation cover problem is far more intriguing for CFDs and SPC views. One way around this is by means of heuristic, at a price: it may not always be able to find a cover.

In contrast, we provide an algorithm to compute a minimal cover of all CFDs propagated via SPC views in the absence of finite-domain attributes, by extending a practical algorithm proposed in [Got87] for computing a cover of FDs propagated via projection views. Despite the increased expressive power of CFDs and SPC views, this algorithm has the same complexity as the algorithm of [Got87]. The algorithm behaves polynomially in many practical cases. Indeed, exponentially large covers are mostly found in examples intentionally constructed. Further, from this algorithm an effective polynomial-time heuristic is immediate: it computes a minimal cover when the cover is not large, and returns a subset of a cover as soon as the computation reaches a predefined bound, when covers are inherently large.

This is the first algorithm for computing minimal propagation covers via SPC views, for FDs or CFDs.

(3) *Experimental study*. We evaluate the scalability of the propagation cover algorithm as well as minimal covers found by the algorithm. We investigate the impact of the number of source CFDs and the complexity of SPC views on the performance of the algorithm. We find that the algorithm is quite efficient; for example, it takes less than 80 seconds to compute minimal propagation covers when given sets of 2000 source CFDs and SPC views with 50 projection attributes and selection conditions defined in terms of the conjunction of 10 domain constraints. Furthermore, it scales well with the number and complexity of source CFDs and SPC views. The minimal covers found by the algorithm are typically small, often containing less CFDs than the sets of input source CFDs. We contend that the algorithm is a promising method for computing minimal propagation covers of CFDs via SPC views, and may find practical use in data integration, data exchange and data cleaning.

This work not only provides the *first* results for CFD propagation, but also *extends the classical results* of FD propagation, an issue that was considered settled 20 years ago, by investigating the propagation problem in the general and practical setting overlooked by prior work. In addition, for both FDs and CFDs, we give the first practical

algorithm for computing minimal propagation covers via SPC views.

Organization. We review CFDs and various fragments of RA in Section 5.2. We establish complexity bounds on dependency propagation in Section 5.3. We provide the algorithm for computing minimal propagation covers via SPC views in Section 5.4. Experimental results are reported in Section 5.5, followed by related work in Section 5.6.

5.2 Dependencies and Views

In this section, we review conditional functional dependencies (CFDs [FGJK08]) and fragments of relational algebra (RA).

5.2.1 Conditional Functional Dependencies

CFDs extend FDs by incorporating a pattern tuple of semantically related data values. In the sequel, for each attribute A in a schema R, we denote its associated domain as $\mathsf{dom}(A)$, which is either infinite (*e.g.,* string, real) or finite (*e.g.,* Boolean, date).

Definition 5.2.1: A CFD φ on a relation schema R is a pair $R(X \to Y,\ t_p)$, where (1) $X \to Y$ is a standard FD, called the FD *embedded in* φ; and (2) t_p is a tuple with attributes in X and Y, referred to as the *pattern tuple* of φ, where for each A in X (or Y), $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed variable '_' that draws values from $\mathsf{dom}(A)$. We separate the X and Y attributes in t_p with '$\|$'.

For CFDs on views (*i.e.,* view CFDs) we also allow a special form $R(A \to B, (x \,\|\, x))$, where A, B are attributes of R and x is a (special) variable. □

Note that traditional FDs are a special case of CFDs, in which the pattern tuples consist of '_' only.

Example 5.2.1: The dependencies we have seen in Section 5.1 can be expressed as CFDs. Some of those are given below:

φ_1: $R([\mathsf{CC}, \mathsf{zip}] \to [\mathsf{street}],\ (44, \_ \,\|\, \_))$,
φ_2: $R([\mathsf{CC}, \mathsf{AC}] \to [\mathsf{city}],\ (44, \_ \,\|\, \_))$,
φ_4: $R([\mathsf{CC}, \mathsf{AC}] \to [\mathsf{city}],\ (44, 20 \,\|\, \text{LDN}))$,
f_1: $R_1(\mathsf{zip} \to \mathsf{street},\ (\_ \,\|\, \_))$.

The standard FD f_1 on source R_1 is expressed as a CFD. □

The semantics of CFDs is defined in terms of a relation $\asymp$ on constants and '_': $\eta_1 \asymp \eta_2$ if either $\eta_1 = \eta_2$, or one of η_1, η_2 is '_'. The operator $\asymp$ naturally extends to

tuples, *e.g.,* (Portland, LDN) $\asymp$ (_, LDN) but (Portland, LDN) $\not\asymp$ (_, NYC). We say that a tuple t_1 *matches* t_2 if $t_1 \asymp t_2$.

An instance D of R *satisfies* $\varphi = R(X \to Y, t_p)$, denoted by $D \models \varphi$, if for *each pair* of tuples t_1, t_2 in D, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$.

Intuitively, φ is a constraint defined on the set $D_\varphi = \{t \mid t \in D, t[X] \asymp t_p[X]\}$ such that for any $t_1, t_2 \in D_\varphi$, if $t_1[X] = t_2[X]$, then (a) $t_1[Y] = t_2[Y]$, and (b) $t_1[Y] \asymp t_p[Y]$. Here (a) enforces the semantics of the embedded FD, and (b) assures the binding between *constants* in $t_p[Y]$ and *constants* in $t_1[Y]$. Note that φ is defined on the subset D_φ of D identified by $t_p[X]$, rather than on the entire D.

An instance D of R satisfies CFD $R(A \to B, (x \parallel x))$ if for any tuple t in D, $t[A] = t[B]$. As will be seen shortly, these CFDs are used to express selection conditions of the form $A = B$ in a view definition, treating domain constraints and CFDs in a uniform framework.

We say that an instance D of a relational schema $\mathcal{R}$ satisfies a set Σ of CFDs defined on $\mathcal{R}$, denoted by $D \models \Sigma$, if $D \models \phi$ for each ϕ in Σ.

Example 5.2.2: Recall the view definition V from Example 5.1.1 and the instances D_1, D_2, D_3 of Fig. 5.1. The view $V(D_1, D_2, D_3)$ satisfies φ_1, φ_2, φ_4 of Example 5.2.1. However, if we remove attribute CC from φ_4, then the view no longer satisfies the modified CFD. Indeed, there are two tuples t_1' and t_5' in $V(D_1, D_2, D_3)$ such that t_1' and t_1 of Fig. 5.1 have identical AC and city values; similarly for t_5 and t_5' of Fig. 5.1. Then t_1' and t_5' violate the modified CFD: they have the same AC attribute but differ in city. □

5.2.2 View Definitions

We study dependency propagation for views expressed in various fragments of RA. It is known that the problem is already undecidable for FDs and views defined in RA [AHV95]. In light of this we shall focus on positive fragments of RA, without set difference, in particular SPC and SPCU.

Consider a relational schema $\mathcal{R} = (S_1, \ldots, S_m)$.

SPC. An SPC query (*a.k.a.* conjunctive query) Q on $\mathcal{R}$ is an RA expression defined in terms of the selection (σ), projection (π), Cartesian product ($\times$) and renaming (ρ) operators. It can be expressed in the normal form below [AHV95]:

$$\pi_Y(R_c \times E_s), \text{ where } E_s = \sigma_F(E_c),\ E_c = R_1 \times \ldots \times R_n,$$

where (a) $R_c = \{(A_1 : a_1, \ldots, A_m : a_m)\}$, a constant relation, such that for each $i \in [1, m]$,

A_i is in Y, A_i's are distinct, and a_i is a constant in $\mathsf{dom}(A_i)$; (b) for each $j \in [1,n]$, R_j is $\rho_j(S)$ for some relation atom in $\mathcal{R}$, and ρ_j is a renaming operator such that the attributes in R_j and R_l are disjoint if $j \neq l$, and A_i does not appear in any R_j; (c) F is a conjunction of equality atoms of the form $A = B$ and $A =$ 'a' for a constant $a \in \mathsf{dom}(A)$.

We also study fragments of SPC, denoted by listing the operators supported: S, P, C, SP, SC, and PC (the renaming operator is included in all these subclasses by default without listing it explicitly). For instance, SC is the class of queries defined with σ, $\times$ and ρ operators.

For example, Q_1 given in Example 5.1.1 can be expressed as a C query: $\{(\mathsf{CC}: 44)\} \times R_1$.

SPCU. SPCU (*a.k.a.* union of conjunctive queries) is an extension of SPC by allowing union ($\cup$). An SPCU query defined on $\mathcal{R}$ can be expressed in normal form $V_1 \cup \ldots \cup V_n$, where V_i's are union-compatible SPC queries. For example, the view V given in Example 5.1.1 is an SPCU query.

In the sequel we only consider SPC and SPCU queries in the normal form, unless stated otherwise.

5.3 Complexity on Dependency Propagation

We now give a full treatment of dependency propagation to CFDs.

Formally, the *dependency propagation problem* is to determine, given a view V defined on a schema $\mathcal{R}$, a set Σ of source dependencies on $\mathcal{R}$, and CFD φ on the view, whether or not φ is *propagated from* Σ *via* V, denoted by $\Sigma \models_V \varphi$, *i.e.,* for any instance D of $\mathcal{R}$, if $D \models \Sigma$ then $V(D) \models \varphi$.

That is, φ is propagated from Σ via V if for any source D that satisfies Σ, the view $V(D)$ is guaranteed to satisfy φ.

We study the problem in a variety of settings. (a) We consider views expressed in various fragments of RA: S, P, C, SP, SC, PC, SPC, SPCU. (b) We study the propagation problem when FDs and CFDs are source dependencies, respectively. We refer to the problem as *propagation from* FD*s to* CFD*s* when the source dependencies are FDs, and as *propagation from* CFD*s to* CFD*s* when the source dependencies are CFDs. (c) We investigate the problem in the absence and in the presence of finite-domain attributes in the schema $\mathcal{R}$, *i.e.,* in the infinite-domain setting and the general setting.

We first study propagation from FDs to CFDs, and then from CFDs to CFDs. Finally, we address a related interesting issue: the emptiness problem for CFDs and views.

5.3.1 Tableaux: A Brief Overview

Tableaux have proved useful in studying relational algebra expressions. Below we briefly review the notion of tableaux of [KP82], which allows multiple rows in the summary.

Consider a relational schema $\mathcal{R} = (R_1, \ldots, R_m)$. A *tableau* T defined on $\mathcal{R}$ is represented as $(\mathsf{Sum}, T_1, \ldots, T_m)$, where for each $i \in [1, m]$, T_i consists of a finite set of free tuples over $R_i(A_1, \ldots, A_k)$, and $k = \mathsf{arity}(R_i)$. For a free tuple $t = (a_1, \ldots, a_k) \in T_i$, a_j is either a constant or a variable for $j \in [1, k]$. If a_j is a constant, then $a_j \in \mathsf{dom}(A_j)$. If a_j is a variable, then $a_j \in \mathsf{Var}$, which is an infinite set of variables. Here Sum is called the *summary*, which also consists of a finite set of free tuples. For a free tuple $s = (w_1, \ldots, w_h) \in \mathsf{Sum}$, where $h = \mathsf{arity}(\mathsf{Sum})$, and for $j \in [1, h]$, w_j is a blank, a constant, or a variable. The variables in Sum are also called distinguished variables, and they must appear in some free tuples from T_i ($1 \leq i \leq m$).

Intuitively, the tableau T represents a query, where T_i's specify a pattern, and the summary tuples represent the tuples to be included in the answer to the query, *i.e.,* those for which the pattern specified by T_i's is found in the database (see, *e.g.,* [AHV95] for detailed discussions about tableau queries). For a database instance D of $\mathcal{R}$, we use $T(D)$ to denote the answer to query T in the database D.

For a tableau query T and a query Q in relational algebra, both defined on the same relational schema $\mathcal{R}$, we say that T and Q are *equivalent* iff for any instance D of $\mathcal{R}$, $T(D) = Q(D)$.

The following is known.

Theorem 5.3.1 *For every SPC expression E, there exists a tableau T such that E is equivalent to T [KP82].*

Note that for an SPC query, one can always find an equivalent tableau query in which the summary consists of a single row [KP82]. Using the translation rules of [SY80, ASU79], it is easy to get the following corollary:

Corollary 5.3.2 *Given an SPC expression E, there is a polynomial time algorithm, which transforms E into an equivalent tableau query.*

5.3.2 Propagation from FDs to CFDs

In the infinite-domain setting, propagation from FDs to FDs has been well studied, *i.e.,* for source FDs and view FDs. It is known that the propagation problem is

- undecidable for views expressed in RA [Klu80], and
- in PTIME for SPCU views [KP82, AHV95].

In this setting, CFDs do not make our lives harder.

Theorem 5.3.3: *In the absence of finite-domain attributes, the dependency propagation problem from* FD*s to* CFD*s is*

- *in* PTIME *for SPCU views, and*
- *undecidable for RA views.* □

Proof: (a) We first show that it is in PTIME for SPCU views. We develop an algorithm for checking propagation, via tableau representations of given SPCU views and view CFDs, by extending the chase technique. We show that the algorithm characterizes propagation and is in PTIME.

More specifically, assume that the set of source dependencies is Σ, the SPCU view V is $e_1 \cup \ldots \cup e_k$, where e_i is an SPC expression for each $i \in [1,k]$, and the view CFD on V is φ. We show that $\Sigma \models_V \varphi$ can be determined in PTIME. We first show the PTIME bound for $k = 1$, and then extend the proof to $k > 1$.

(a.1) We prove that for $k = 1$, $\Sigma \models_V \varphi$ is in PTIME, *i.e.,* $\Sigma \models_V \varphi$ can be determined in PTIME when V is an SPC expression.

To do this, we first give tableau representations of the view FD and the SPC view. We also construct a representation of an source instance, in which tuples contain variables. We then present an extension of Chase defined on this instance. Finally, we show that the Chase process is actually a PTIME algorithm for testing propagation, *i.e.,* $\Sigma \models_V \varphi$ iff the chase process terminates on the instance and moreover, it either yields an empty view or leads to a view that satisfies the view dependency φ. Further, the chase process is in PTIME.

Assume that the view $V(B_1,\ldots,B_n)$ be defined on h source relations: $R_1(A_{11},\ldots,A_{1n_1})$, ..., $R_h(A_{h1},\ldots,A_{hn_h})$. Let the view dependency CFD φ be $V(B_{i_1}\ldots B_{i_g} \rightarrow B,\ t_p)$. Their corresponding tableau representations T_φ and T_V are shown in Fig. 5.2. In tableau T_φ, if $t_p[B_{i_j}]$ ($j \in [1,g]$) is a constant, then $x_j = t_p[B_{i_j}]$. Otherwise, x_j is a new variable distinct from x_l for $l \in [1, j-1]$. Further, y_1 and y_2 are two new variables distinct from x_j ($j \in [1,g]$). In tableau T_V, for each $j \in [1,n]$, v_{sj} appearing in Sum is a blank, a constant, or some variable appearing in the free tuples for R_i ($i \in [1,h]$).

Note that the CFD φ is defined on the view, and thus B and each B_{i_j} of T_φ are attributes in $\{B_1,\ldots,B_n\}$ of the summary tuple of T_V, for each $j \in [1,g]$.

B_{i_1}	...	B_{i_g}	B
x_1	...	x_g	y_1
x_1	...	x_g	y_2
$y_1 = y_2 \asymp t_p[B]$			

(a) T_φ for $\varphi = V(B_{i_1} \ldots B_{i_g} \rightarrow B,\ t_p)$

Sum		
B_1	...	B_n
v_{s1}	...	v_{sn}

$R_i(A_{i1}, \ldots, A_{in_i})$ for $i \in [1,h]$

A_{i1}	...	A_{in_i}
v_{m_i+1}	...	$v_{m_i+n_i}$
$v_{m_i+n_i+1}$	...	$v_{m_i+2*n_i}$
	$\vdots$	
$v_{m_i+p_i*n_i+1}$	...	$v_{m_i+(p_i+1)*n_i}$

(b) T_V for $V(B_1, \ldots, B_n)$

$R_i(A_{i1}, \ldots, A_{in_i})$ for $i \in [1,h]$

A_{i1}	...	A_{in_i}
$\rho_1(v_{m_i+1})$	...	$\rho_1(v_{m_i+n_i})$
	$\vdots$	
$\rho_1(v_{m_i+p_i*n_i+1})$	...	$\rho_1(v_{m_i+(p_1+1)*n_i})$
$\rho_2(v_{m_i+1})$	...	$\rho_2(v_{m_i+n_i})$
	$\vdots$	
$\rho_2(v_{m_i+p_i*n_i+1})$	...	$\rho_2(v_{m_i+(p_i+1)*n_i})$

(c) Instance I

/* Here $m_i = \Sigma_{j=1}^{i-1}(p_{j-1}+1) * n_{j-1}$ in (b) and (c) */

Figure 5.2: Tableau Representations

In order to test whether φ holds on V or not, it suffices to check whether there exist two tuples $t_1, t_2 \in V$ such that either $t_1[B] \neq t_2[B]$ or $t_1[B] \not\asymp t_p[B]$ while $t_1[B_{i_1} \ldots B_{i_g}] = t_2[B_{i_1} \ldots B_{i_g}] \asymp t_p[B_{i_1} \ldots B_{i_g}]$. To check the existence of t_1, t_2 in V, we construct tuples in source relations shown in Fig. 5.2, along with two mappings ρ_1 and ρ_2 from constants

and variables of T_V to those of T_φ, such that if t_1, t_2 exist, then t_1 is generated by V applied to those source tuples given by ρ_1, and t_2 is generated by V and ρ_2.

Intuitively, ρ_1 and ρ_2 aim to map the summary tuple $(v_{s1}, \ldots, v_{sn})$ of T_V to $(x_1, \ldots, x_k, y_1)$ and $(x_1, \ldots, x_k, y_2)$ of T_φ respectively. We define ρ_1 and ρ_2 based on the correspondences between the attributes in T_V and those in T_φ (*i.e.,* v_{si} and x_l), as follows. We start with the definition of ρ_1.

Case 1. When $B_j = B_{i_l}$ for some $1 \leq j \leq n$ and $1 \leq l \leq g$, *i.e.,* the attribute B_{i_l} in T_φ corresponds to B_j in the summary of T_V, v_{sj} should be mapped to x_l. We define ρ_1 as follows. (a) If v_{sj} is a variable, then let $\rho_1(v_{sj}) = x_l$. (b) If both v_{sj} and x_l are the same constant, then let $\rho_1(v_{sj}) = v_{sj}$. (c) If both v_{sj} and x_l are constants but $v_{sj} \neq x_l$, then $\rho_1(v_{sj})$ is *undefined*.

Case 2. When $B_j = B$ for some $1 \leq j \leq n$, v_{sj} should be mapped to y_1. More specifically, (a) if v_{sj} is a variable, then let $\rho_1(v_{sj}) = y_1$. (b) If v_{sj} is a constant and $v_{sj} \asymp t_p[B]$, then let $\rho_1(v_{sj}) = v_{sj}$. Otherwise, let $\rho_1(v_{sj})$ be *undefined*.

Case 3. Otherwise, B_j does not correspond to any attribute in T_φ. Then we let $\rho_1(v_j) = v_j$. We also let $\rho_1(v_j) = v_j$ for $n \leq j \leq (m_h + (p_h + 1) * n_h)$.

Similarly, we define the mapping ρ_2 to map the summary tuple $(v_{s1}, \ldots, v_{sn})$ of T_V to $(x_1, \ldots, x_k, y_2)$ of T_φ. Again, we consider three cases.

Cases 1 and 2. Here ρ_2 is defined along the same lines as ρ_1.

Case 3. For each constant v_j in T_V that is not covered by Case 1 or 2, where $1 \leq j \leq (m_h + (p_h + 1) * n_h)$, let $\rho_2(v_j) = v_j$.

Case 4. For all those variables v_{j_1}, v_{j_2} in T_V that are not covered by Case 1 or 2, where $1 \leq j_1, j_2 \leq (m_h + (p_h + 1) * n_h)$, we define ρ_2 such that

- $\rho_2(v_{j_1}) \neq v_{j_1}$, $\rho_2(v_{j_2}) \neq v_{j_2}$;
- $\rho_2(v_{j_1}) \neq \rho_2(v_{j_2})$ if $v_{j_1} \neq v_{j_2}$;
- there does not exist any variable v_j in T_V such that $\rho_2(v_{j_1}) = v_j$ or $\rho_2(v_{j_2}) = v_j$.

That is, ρ_2 uses variables different from ρ_1 in this case. It is easy to see that such ρ_2 exists.

If there exists a constant v_j in T_V such that $\rho_1(v_j)$ (resp. $\rho_2(v_j)$) is *undefined*, we say ρ_1 (resp. ρ_2) is *undefined*. If ρ_1 or ρ_2 is *undefined*, it is easy to verify that no tuples in V match the LHS of φ. From this it follows that $\Sigma \models_V \varphi$. If both ρ_1 and ρ_2 are well defined, we create an instance I of the source relation schemas $(R_1, \ldots, R_h)$, such that the instance I_i of R_i is $\rho_1(T_V.R_i) \cup \rho_2(T_V.R_i)$ (shown in Fig. 5.2 (*c*)) for $i \in [1, h]$.

Chase. Next, we show how to check whether $\Sigma \models_V \varphi$ by performing Chase on the instance I. Without loss of generality, assume a total order $\leq$ on the variables in I.

Consider an FD $\phi = R_i(X \rightarrow Y)$ in Σ, where $1 \leq i \leq h$. We define the chase of I by ϕ, denoted by Chase_ϕ, as follows. For any tuples $t,\ t' \in I_i$ of I, if $t[X] = t'[X]$ and $t[A] \neq t'[A]$ for some $A \in Y$, then Chase_ϕ applies ϕ to I as follows. (a) If both $t[A]$ and $t'[A]$ are variables, then let $t[A] = t'[A]$ if $t'[A] \leq t[A]$. Otherwise, let $t'[A] = t[A]$. (b) If $t[A]$ is a constant and $t'[A]$ is a variable, then let $t'[A] = t[A]$. (c) If $t[A]$ is a variable and $t'[A]$ is a constant, then let $t[A] = t'[A]$. (d) If both $t[A]$ and $t'[A]$ are (distinct) constants, Chase_ϕ is *undefined*.

Here by $t[A] = t'[A]$ (resp. $t'[A] = t[A]$) means replacing each appearance of $t[A]$ (resp. $t'[A]$) in I with $t'[A]$ (resp. $t[A]$).

This process repeats for all FDs in Σ until one of the following cases happens: (1) no further changes can be incurred to I, (2) the Chase process is *undefined*, or (3) y_1 and y_2 are identified during the process.

Observe that Chase process must terminate. Indeed, there are at most $2 * (m_h + (p_h + 1) * n_h)$ variables in I. Let $|I| = 2 * (m_h + (p_h + 1) * n_h)$, then there are at most $O(|I|^2)$ value assignments, and each can be done in $O(|\Sigma||I|^2)$ time. From this, it follows that the Chase process is in $O(|\Sigma||T|^4)$ time.

We finally show that the chase process suffices to determine whether or not $\Sigma \models_V \varphi$. Consider the following cases. (a) If the Chase process is *undefined*, then the view V must be empty. As a result, obviously $\Sigma \models_V \varphi$. (b) If the *chase* terminates due to $y_1 = y_2$, then $\Sigma \models_V \varphi$ since for any two tuples $t_1, t_2 \in V$, we have that $t_1[B] = t_2[B] \asymp t_p[B]$, if $t_1[B_{i_1} \ldots B_{i_g}] = t_2[B_{i_1} \ldots B_{i_g}] \asymp t_p[B_{i_1} \ldots B_{i_g}]$. (c) Otherwise, the chase of I by FDs yields a counterexample for the propagation, and thus $\Sigma \not\models_V \varphi$. That is, $\Sigma \models_V \varphi$ if and only if the process terminates with cases (a) or (b).

(1) We first show that if $\Sigma \models_V \varphi$, then the process terminates with cases (a) or (b). This is easy to verify since case (c) produces a counterexample for the propagation.

(2) We then show that if $\Sigma \not\models_V \varphi$, then the process terminates with cases (c). Since $\Sigma \not\models_V \varphi$, there exists a database $D = (I_1, \ldots, I_h)$ such that $V(D) \models \Sigma$, but $V(D) \not\models \varphi$. That is, there exist two tuples $t_1, t_2 \in V(D)$ such that $t_1[B_{i_1}, \ldots, B_{i_g}] = t_2[B_{i_1}, \ldots, B_{i_g}] \asymp t_p[B_{i_1}, \ldots, B_{i_g}]$, but not $t_1[B] = t_2[B] \asymp t_p[B]$. Let $D' = (I'_1, \ldots, I'_h)$ such that for each $i \in [1, h]$, (a) $I'_i \subseteq I_i$ and (b) all tuples in I'_i contribute to the existences of tuples t_1 and t_2 in $V(D)$. It is easy to see that for each $i \in [1, h]$ I'_i can be embedded into the instance constructed by the two mappings ρ_1 and ρ_2 described before. If the process terminates with cases (a) or (b), then $V(D') \models \varphi$. This contradicts with the assumption

that $V(D') \not\models \varphi$.

From these it follows that the Chase process is a PTIME algorithm that determines whether $\Sigma \models_V \varphi$, when V is an SPC expression.

(a.2) We next prove that for $k > 1$, $\Sigma \models_V \varphi$ can also be determined in PTIME.

When V is $e_1 \cup \ldots \cup e_k$, V can be represented as a set of tableaux $T = (T_1, \ldots, T_k)$, where T_i is the tableau representation of SPC expression e_i for $i \in [1,k]$. Here to construct source relations such that there exists a pair of view tuples t_1 and t_2 in V that serves as a counterexample for $\Sigma \models_V \varphi$, we have to consider t_1 and t_2 generated by any two of the k SPC expressions, namely, t_1 may be generated via e_i and t_2 may be produced by e_j while i and j are not necessarily the same. In total, there are k^2 combinations of the SPC expressions, and the checking needs to be performed on each of these combination. The Chase given above for $k = 1$ can be easily extended to check each combination. From these it follows that there is a PTIME algorithm to check whether $\Sigma \models_V \varphi$.

(b) The undecidability follows from its counterpart for FDs (see [AHV95]), since CFDs subsume FDs. □

From Theorem 5.3.3 it follows immediately that propagation from FDs to CFDs is also in PTIME for views expressed in fragments of SPCU, *e.g.,* SPC, SP, SC and PC views.

In the general setting, *i.e.,* when finite-domain attributes may be present, the propagation analysis becomes harder. Below we show that even for propagation from FDs to FDs and for simple SC views, the problem is already intractable.

Theorem 5.3.4: *In the general setting, the dependency propagation problem from* FD*s to* FD*s is co*NP*-complete for SC views.* □

Proof: We first present an NP algorithm that, given source FDs Σ, a view FD φ and an SC view V, decides whether $\Sigma \not\models_V \varphi$ or not. The algorithm is based on tableaux and instance constructed in the same way as in the proof of Theorem 5.3.3. The difference here is that we have to deal with variables that are associated with finite-domain attributes, such that all those variables are instantiated with constants from their corresponding finite domains. There are exponentially many such instantiations that need to be checked. To cope with this, we first guess an instantiation, and then check, *w.r.t.* this instantiation, whether or not $\Sigma \not\models_V \varphi$. The latter can be done in polynomial time by using the PTIME algorithm given in the proof of Theorem 5.3.3. Note that $\Sigma \not\models_V \varphi$ if and only if there exists an instantiation such that $\Sigma \not\models_V \varphi$ *w.r.t.* the instantiation. From

this it follows that this problem is in coNP.

The lower bound is shown by reduction from 3SAT to the complement of the propagation problem (*i.e.,* the problem to decide whether $\Sigma \not\models_V \varphi$), where 3SAT is NP-complete (cf. [GJ79]). More specifically, consider an instance $\phi = C_1 \wedge \cdots \wedge C_n$ of 3SAT, where all the variables in ϕ are $x_1, \ldots, x_m$, C_j is of the form $y_{k_1} \vee y_{k_2} \vee y_{k_3}$, and moreover, for $i \in [1,3]$, y_{k_i} is either $x_{p_{ki}}$ or $\overline{x_{p_{ki}}}$, for $p_{ki} \in [1,m]$. We shall construct a database schema $\mathcal{R} = (R_0, R_1, \ldots, R_n)$, a set Σ of FDs on $\mathcal{R}$, a view V defined by an SC expression, and a FD φ on V. Then we show that ϕ is satisfiable if and only if $\Sigma \not\models_V \varphi$.

Let the schema of R_0 be (X, A, Z), where $\mathsf{Dom}(X) = \{1, \ldots, m, \ldots\}$ (*e.g.,* int), $\mathsf{Dom}(A) = \mathsf{Dom}(Z) = \{0, 1\}$, *i.e.,* A and Z are finite-domain attributes. Moreover, an FD $\varphi_0 = R_0(X \to A)$ is defined on R_0. Intuitively, for each tuple $t \in R_0$, $t[X]$, $t[A]$ and $t[Z]$ encode a variable, its corresponding truth assignment, and the truth value of ϕ, respectively. The FD φ_0 guarantees that each variable has a unique truth assignment.

Let the schema of R_i be (A_1, A_2, X^i, A^i) for $i \in [1,n]$, where $\mathsf{Dom}(A_1) = \mathsf{Dom}(A_2) = \mathsf{Dom}(A^i) = \{0, 1\}$, and $\mathsf{Dom}(X^i) = \{1, \ldots, m, \ldots\}$ (*e.g.,* int), *i.e.,* A_1, A_2 and A are finite-domain attributes. Moreover, we define two FDs $\varphi_{i_1} = R_i(A_1, A_2 \to X^i A^i)$, and $\varphi_{i_2} = R_i(X^i \to A^i)$ on R_i. Intuitively, for each tuple $t \in R_i$, $t[X^i]$ and $t[A^i]$ is to encode a variable and its corresponding truth assignment that make the clause C_i true. Here A_1 and A_2 serve as a "counter" to assure that there are three variables that could satisfy C_i (note that while A_1 and A_2 encode numbers from 0 to 3, one of these is redundant as C_i is defined in terms of 3 literals). The FD φ_{i_1} assures that A_1 and A_2 are used as a "key" such that only variables in C_i and their appropriate truth assignments are considered. Again, the FD φ_{i_2} assures that each variable has a unique truth assignment.

Let the set Σ of source FDs be $\{\varphi_0, \varphi_{1_1}, \varphi_{1_2}, \ldots, \varphi_{n_1}, \varphi_{n_2}\}$.

We next define the SC expression V to be $e \times e_{0_1} \times e_{0_2} \times e_1 \times \ldots \times e_n$, where

- $e = R_0$,
- $e_{0_1} = \sigma_{X=1}(R_0) \times \ldots \times \sigma_{X=m}(R_0)$,
- $e_{0_2} = \sigma_{R_0.X=R_1.X^1 \wedge R_0.A=R_1.A^1}(R_0 \times R_1) \times \ldots \times$
  $\sigma_{R_0.X=R_n.X^n \wedge R_0.A=R_n.A^n}(R_0 \times R_n)$, and
- $e_j = \sigma_{X^j=p_{k1} \wedge A^j=a_1 \wedge A_1=0 \wedge A_2=0}(R_j) \times \sigma_{X^j=p_{k2} \wedge A^j=a_2 \wedge A_1=0 \wedge A_2=1}(R_j) \times$
  $\sigma_{X^j=p_{k3} \wedge A^j=a_3 \wedge A_1=1 \wedge A_2=0}(R_j) \times \sigma_{X^j=p_{k1} \wedge A^j=a_1 \wedge A_1=1 \wedge A_2=1}(R_j)$,
  where $j \in [1,n]$, and for $i \in \{1,2,3\}$, $a_i = 1$ if $y_{ki} = x_{p_{ki}}$ and $a_i = 0$ if $y_{ki} = \overline{x_{p_{ki}}}$.

Intuitively, (a) the subexpression e_{0_1} assures that R_0 contains a truth assignment for all variables of ϕ, *i.e.,* $x_1, \ldots, x_m$ appear in e_{0_1}; (b) e_{0_2} is to assure that the truth assignments of variables in R_0 and the truth assignments of variables in R_j are consistent, for $j \in$

$[1,n]$; and (c) for each $j \in [1,n]$, e_j encodes the clause C_j: it enumerates all the truth assignments that make C_j true (note that when $A_1 = 1$ and $A_2 = 1$, the truth assignment is redundant: it repeats the assignment when $A_1 = 0$ and $A_2 = 0$).

Based on the definition of V and Σ, it is easy to verify the following: for any instance I of $(R_0, R_1, \ldots, R_n)$, if $V(I)$ is not empty then the instance of R_0 in I contains a truth assignment that makes ϕ true.

Finally, we define φ to be $V(X, A \to Z)$, where attributes X, A, Z come from the subexpression e.

Now we verify that ϕ is satisfiable if and only if the view FD φ is not propagated from the source FDs Σ via V.

Suppose that $\Sigma \not\models_V \varphi$. Then there exists an instance I such that $I \models \Sigma$, while $V(I) \not\models \varphi$. Since $V(I) \not\models \varphi$, we have that $V(I)$ is not empty. As remarked earlier, the instance of R_0 in I contains a truth assignment that makes ϕ true. Thus ϕ is satisfiable.

Conversely, if ϕ is satisfiable, then there is a truth assignment ξ that makes ϕ true. We construct an instance I of $(R_0, R_1, \ldots, R_n)$ such that $I \models \Sigma$ but $V(I) \not\models \varphi$. Let the instance of R_0 in I consist of $2m$ tuples $\{t_1, \ldots, t_{2m}\}$. For each $i \in [1, \ldots, m]$, let (a) $t_i[X] = i$, $t_i[A] = \xi(x_i)$ and $t_i[Z] = 0$; and (b) $t_{m+i}[X] = i$, $t_{m+i}[A] = \xi(x_i)$ and $t_{m+i}[Z] = 1$. That is, t_i and t_{m+i} agree on their X and A attributes, *i.e.,* they encode the same truth assignment for X_i; however, they differ in their Z attributes. For $j \in [1,n]$, let the instance of R_j in I contain exactly four tuples as specified by e_j, which are all true assignments that make C_j true only. It is easy to verify that $I \models \Sigma$ and $V(I) \not\models \varphi$. From this it follows that φ is not propagated from Σ via V.

Therefore, the problem is coNP-complete. □

In contrast to the PTIME bound in the infinite-domain setting [KP82, AHV95], Theorem 5.3.4 shows that the presence of finite-domain attributes does complicate the analysis of propagation from FDs to FDs, and should be thoroughly studied.

Theorem 5.3.5: *In the general setting, the dependency propagation problem from* FD*s to* CFD*s is*

- *in* PTIME *for PC views,*
- *in* PTIME *for SP views,*
- *co*NP*-complete for SC views,*
- *co*NP*-complete for SPCU views,*
- *undecidable for RA views.*

□

Proof: (a) We develop a PTIME algorithm that, given source FDs Σ, a view CFD φ and an PC view V, checks whether $\Sigma \models_V \varphi$ or not. In fact the PTIME algorithm given in the proof of Theorem 5.3.3 suffices here. To see this, assume that V is $\pi_{B_1,\ldots,B_m}(R_1 \times \ldots \times R_n)$, and that φ is $V(B_{i_1} \ldots B_{i_g} \rightarrow B, t_p)$, where $B_{i_j}, B \in \{B_1, \ldots, B_m\}$ for $j \in [1, g]$. It is easy to verify that if $\Sigma \models_V \varphi$, all of $B_{i_1} \ldots B_{i_g}$ and B must be attributes of the same relation R_i $(1 \leq i \leq n)$. Therefore, without loss of generality, we assume that there is only one relation R involved in the PC view V. We construct an instance I as in the proof of Theorem 5.3.3. In this case, I only need to contain two tuples of R. As a result, the instantiations of finite-domain variables are not necessary because each domain has at least two elements: we can simply construct the two tuples with distinct values whenever necessary, regardless of what values they are. Then the algorithm given in the proof of Theorem 5.3.3 suffices to determine whether or not $\Sigma \models_V \varphi$.

(b) Similar to the proof of (a), one can verify that the algorithm given in the proof of Theorem 5.3.3 suffices to determine whether $\Sigma \models_V \varphi$ or not in this case. Indeed, when V is defined as an SP expression, only one source relation is involved in V. Then the argument for (a) suffices to show that the algorithm in the proof of Theorem 5.3.3 is a PTIME algorithm for determining the propagation in this case.

(c, d) We first show the dependency propagation problem from FDs to CFDs is in coNP for SPCU views. Then we show the dependency propagation problem from FDs to CFDs is in coNP-hard for SC views.

The coNP upper bound is verified by giving an NP algorithm for deciding $\Sigma \not\models_V \varphi$ for any given source FDs Σ, view CFD φ and SPCU view V. The proof is similar to that of Theorem 5.3.4. The only difference here is that we represent the CFD as a tableau, instead of an FD. Again we have deal with variables associated with finite-domain attributes, such that all those variables are instantiated with constants from their corresponding finite domains. There are exponentially many such instantiations that need to be checked. To cope with this, we first guess an instantiation, and then check, *w.r.t.* this instantiation, whether or not $\Sigma \not\models_V \varphi$. The latter can be done in polynomial time by using the PTIME algorithm given in the proof of Theorem 5.3.3. Note that $\Sigma \not\models_V \varphi$ if and only if there exists an instantiation such that $\Sigma \not\models_V \varphi$ *w.r.t.* the instantiation. From this it follows that this problem is in coNP.

The coNP lower bound follows from the proof of Theorem 5.3.4 for SC views, since CFDs subsume FDs.

(e) The undecidability follows from Theorem 5.3.3, since the general setting subsumes the infinite-domain setting. □

Theorem 5.3.5 also tells us that in the general setting, propagation from FDs to CFDs is (a) in PTIME for S, P and C views, and (b) coNP-complete for SPC views.

In addition, since FDs are a special case of CFDs, Theorems 5.3.4 and 5.3.5 together yield a complete picture of complexity bounds on the dependency propagation problem from FDs to FDs in the general setting.

Corollary 5.3.6: *In the general setting, the propagation problem from* FD*s to* FD*s is in* PTIME *for SP and PC views, and is co*NP*-complete for SPC and SPCU views.* □

5.3.3 Propagation from CFDs to CFDs

Upgrading source dependencies from FDs to CFDs does not incur extra complexity for propagation analysis, in the infinite-domain setting. That is, the bounds of Theorem 5.3.3 remain intact, which are the same as for FD propagation.

Theorem 5.3.7: *In the absence of finite-domain attributes, the dependency propagation problem from* CFD*s to* CFD*s is*

- *in* PTIME *for SPCU views, and*
- *undecidable for RA views.* □

Proof: (a) The PTIME bound is again verified by developing a polynomial time checking algorithm. The algorithm is similar to that given in the proof of Theorem 5.3.3. The only difference here is that a minor extension of the chase rules is used here to cope with CFDs instead of FDs.

(b) The undecidability follows from Theorem 5.3.3, since FDs are a special case of CFDs. □

This tells us that propagation from CFDs to CFDs is also in PTIME for SPC, SP, SC and PC views.

When it comes to the general setting, however, the problem becomes intractable even for very simple views.

Corollary 5.3.8: *In general setting, the dependency propagation problem from* CFD*s to* CFD*s is*

- *co*NP*-complete for views expressed as S, P or C queries;*
- *co*NP*-complete for SPCU views; and*
- *undecidable for RA views.*

□

Proof: (a, b) We first show that the propagation problem is coNP-hard for views expressed as S, P, or C queries. Then we show the propagation problem for SPCU views is in coNP.

We show the lower bound by reduction from the implication problem for CFDs. The *implication problem* for CFDs is to determine, given a set Σ of CFDs and a single CFD φ defined on a relation schema R, whether or not Σ entails φ, denoted by $\Sigma \models \varphi$, *i.e.,* whether or not for all instances I of R, if $I \models \Sigma$ then $I \models \varphi$. It is known that in the presence of finite-domain attributes, CFD implication is already coNP-complete [FGJK08]. Observe that the implication problem for CFDs is a special case of the dependency propagation problem by limiting the views to the identity mappings, which are expressible as S, P, C or SPCU queries. From these it follows that the propagation problem is coNP-hard for views expressed as S, P or C queries.

The coNP upper bound is verified along the same lines as the proof of Theorem 5.3.5 for SPCU views. The only difference here is that we need to extend chase to deal with source CFDs instead of source FDs.

(c) The undecidability follows from Theorem 5.3.5, since FDs are a special case of CFDs. □

From Corollary 5.3.8 it follows that the propagation problem is also coNP-complete for SC, PC, SP and SPC views.

5.3.4 Interaction between CFDs and Views

An interesting aspect related to dependency propagation is the interaction between source CFDs and views.

Example 5.3.1: Consider a CFD $\phi = R(A \rightarrow B,\ (\_ \parallel b_1))$ defined on a source $R(A,B,C)$, and an S view $V = \sigma_{B=b_2}(R)$, with $b_2 \neq b_1$. Then for any instance D of R that satisfies ϕ, $V(D)$ is always *empty*. Indeed, the CFD ϕ assures that the B attributes of all tuples t in D have the same constant: $t[B] = b_1$, no matter what value $t[A]$ has. Hence V cannot possibly find a tuple t in D that satisfies the selection condition $B = b_2$. As a result, any source CFDs are "propagated" to the view, since the view satisfies any CFDs. □

This suggests that we consider the *emptiness problem for views and* CFD*s*: it is to determine, given a view V defined on a schema $\mathcal{R}$ and a set Σ of CFDs on $\mathcal{R}$, whether or not $V(D)$ is always empty for all instances D of $\mathcal{R}$ where $D \models \Sigma$.

It turns out that this problem is nontrivial.

Theorem 5.3.9: *In the general setting, the emptiness problem is co*NP*-complete for* CFD*s and SPCU views.* □

Proof: To do this, we first show that the emptiness problem for CFDs and SPCU views is coNP-hard. We then show that this problem is in coNP.

The lower bound is verified by reduction from the satisfiability problem for CFDs to the complement of the emptiness problem. The *satisfiability problem* for CFDs is to determine, given a set Σ of CFDs defined on a relation schema R, whether or not there exists a nonempty instance I of R such that $I \models \Sigma$. It is known that the satisfiability problem for CFDs is NP-complete [FGJK08]. Obviously the satisfiability problem is a special case of the complement of the emptiness problem (*i.e.,* the non-emptiness problem), where views are defined as the identity mapping. Putting these together, we have that the emptiness problem is coNP-hard.

The upper bound is verified by providing an NP algorithm to check the non-emptiness based on an extended Chase process. More specifically, assume that the SPCU expression V is $e_1 \cup \ldots \cup e_k$, where for each $i \in [1,k]$, e_i is an SPC expression. Note that if V is not empty, then there must exist at least one e_i ($1 \leq i \leq k$) such that e_i is not empty.

To present the NP algorithm, we first give a tableau representation T_{e_i} for each SPC view e_i. We then extend the Chase technique to handle T_{e_i}. Finally, we show that the Chase process is actually an NP algorithm for checking the non-emptiness.

The construction of tableau T_{e_i} follows the same way as that of T_V given in the proof of Theorem 5.3.3. Without loss of generality, we assume that there is a total order $\leq$ on the variables in T_{e_i}. Variables in T_{e_i} that are associated with finite-domain attributes are instantiated with constants from their corresponding finite domains. There are exponentially many such instantiations. To cope with this, we first guess an instantiation; we then check, *w.r.t.* this instantiation, whether or not V yields a non-empty instance on some source database. To show that the algorithm is in NP, it suffices to show that the checking step can be done in PTIME, for each instantiation.

For each instantiation of tableau T_{e_i}, the following Chase process is performed to apply each CFD $\varphi = R(X \to A, t_p)$ in Σ. We next extend the chase rules for CFDs. There are two cases to consider, depending on $t_p[A]$.

Case 1. The pattern value $t_p[A]$ is an unnamed variable '_'. In this case, for any free R tuples t, t' of T_{e_i}, if $t[X] = t'[X]$, $t[X] \asymp t_p[X]$ but $t[A] \neq t'[A]$, we do the following. (a) If

both $t[A]$ and $t'[A]$ are variables, then we let $t[A] = t'[A]$ if $t'[A] \leq t[A]$, or $t'[A] = t[A]$ if $t[A] \leq t'[A]$. (b) If $t[A]$ is a constant and $t'[A]$ is a variable, then let $t'[A] = t[A]$. (c) If $t[A]$ is a variable and $t'[A]$ is a constant, then let $t[A] = t'[A]$. (d) If both $t[A]$ and $t'[A]$ are (distinct) constants, then we say that the Chase process is *undefined*.

Case 2. The pattern value $t_p[A]$ is a constant. In this case, for any free R tuple t of T_{e_i}, if $t[X] \asymp t_p[X]$ and $t[A] \neq t_p[A]$, we do the following. (a) If $t[A]$ is a variable, then we let $t[A] = t_p[A]$. (b) If $t[A]$ is a (distinct) constant, then we say that the Chase process is *undefined*.

Here by $t[A] = t'[A]$ (resp. $t'[A] = t[A]$, $t[A] = t_p[A]$) we mean replacing each appearance of $t[A]$ (resp. $t'[A]$, $t[A]$) in T_{e_i} with $t'[A]$ (resp. $t[A]$, $t_p[A]$).

This process repeats for all source CFDs in Σ until either no further changes can be incurred to T_{e_i}, or the Chase process becomes *undefined* for some CFD. It is easy to see that one of these cases must happen.

We now show that the Chase process actually yields an algorithm for checking the non-emptiness. If the Chase process terminates because no changes can be made to T_{e_i}, we can easily construct a source instance I such that $I \models \Sigma$ and $V(I)$ is non-empty. Indeed, the instance I can be constructed by instantiating variables in the final chasing result of T_{e_i} with pairwise different constants. If the Chase process terminates because it becomes *undefined* for some CFD, then the view must be empty *w.r.t.* the instantiation. From these it follows that e_i is non-empty if and only if there exists an instantiation such that the Chase process terminates because of no changes.

Finally we verify that the Chase process above is in NP. It suffices to show that for each instantiation, the Chase process for T_{e_i} terminates in PTIME. Without loss of generality, we assume that T_{e_i} is the same as T_V shown in Fig. 5.2. There are at most $(m_h + (p_h + 1) * n_h)$ variables in T_{e_i}. Let $|T_{e_i}| = (m_h + (p_h + 1) * n_h)$, then there are at most $O(|T_{e_i}|^2)$ value assignments, and each can be done in $O(|\Sigma||T_{e_i}|^2)$ time. Thus the Chase process must terminate in $O(|\Sigma||T_{e_i}|^4)$ time.

Putting these together, we have that the emptiness problem is coNP-complete for CFDs and SPCU views. □

The lower bound is not surprising: it is already NP-hard for deciding whether there exists a nonempty database that satisfies a given set of CFDs, in the general setting [FGJK08]. This, known as the consistency problem [FGJK08], is a special case of the complement of the emptiness problem when the view is the identity mapping. Theorem 5.3.9 tells us that adding views does not make our lives harder: the NP upper

bound for the consistency problem for CFDs remains intact.

In the absence of finite-domain attributes, the implication problem for CFDs becomes tractable [FGJK08]. It is also the case for the emptiness problem for SPCU views and CFDs: a PTIME algorithm can be developed for the emptiness test.

Theorem 5.3.10: *Without finite-domain attributes, the emptiness problem is in* PTIME *for* CFD*s and SPCU views.* □

Proof Sketch: In the absence of finite-domain attributes, one can readily turn the NP algorithm given in the proof of Theorem 5.3.9 into a PTIME one, since instantiation of finite-domain attributes, which would require a nondeterministic guess, is no longer needed here. □

5.4 Computing Covers of View Dependencies

We have studied dependency propagation in Section 5.3 for determining whether a *given* view CFD is propagated from source CFDs (or FDs). In this section we move on to a related yet more challenging problem, referred to as the propagation cover problem, for finding a minimal cover of *all* view CFDs propagated from source CFDs. As remarked in Section 5.1, this problem is important for data integration and data cleaning, among other things. Furthermore, an algorithm for finding a propagation cover also readily provides a solution to determining whether a given CFD ϕ is propagated from a given set Σ of source CFDs via an SPC view V: one can simply compute a minimal cover Γ of all CFDs propagated from Σ via V, and then check whether Γ implies ϕ, where CFD implication is already studied in [FGJK08].

No matter how important, this problem is hard. As will be seen soon, most prior algorithms for finding FD propagation covers always take exponential time.

The main result of this section is an algorithm for computing a minimal cover of all view CFDs propagated from source CFDs, via SPC views. It is an extension of a practical algorithm proposed in [Got87], for computing covers of FDs propagated via projection views. It has the same complexity as that of [Got87], and behaves polynomially in many practical cases. It also yields an algorithm for computing propagation covers when FDs are source dependencies, a special case.

To simplify the discussion we assume the absence of finite-domain attributes, the same setting as the classical work on FD propagation [AHV95, Got87, Klu80, KP82]. In this setting, the emptiness problem for CFDs and SPC views, and the CFD propaga-

tion problem via SPC views are all in PTIME. We consider *w.l.o.g.* CFDs in the normal form: $(R : X \to A, t_p)$, where A is a single attribute. Indeed, each CFD of the general form given in Section 5.2 can be converted in linear time to an equivalent set of CFDs in the normal form [FGJK08].

In the rest of the section, we first state the propagation cover problem and discuss its challenges in Section 5.4.1. We then provide basic results in Section 5.4.2, on which our algorithm is based. The algorithm is presented in Section 5.4.3.

5.4.1 The Propagation Cover Problem

Implication and cover. We say that a set Σ of CFDs defined on a schema $\mathcal{R}$ *implies* another CFD φ on $\mathcal{R}$, denoted by $\Sigma \models \varphi$, if for any instance D of $\mathcal{R}$, if $D \models \Sigma$ then $D \models \varphi$.

A *cover* of a set Σ of CFDs is a subset Σ_c of Σ such that for each CFD φ in Σ, $\Sigma_c \models \varphi$. In other words, Σ_c is contained in, and is equivalent to, Σ. For a familiar example, recall the notion of the closure F^+ of a set F of FDs, which is needed for designing normal forms of relational schema (see, *e.g.,* [AHV95]). Then F is a cover of F^+.

A *minimal cover* Σ_{mc} of Σ is a cover of Σ such that

- no proper subset of Σ_{mc} is a cover of Σ, and
- for each CFD $\phi = R(X \to A,\ t_p)$ in Σ_{mc}, there exists no proper subset $Z \subset X$ such that $(\Sigma_{mc} \cup \{\phi'\}) - \{\phi\} \models \phi$, where $\phi' = R(Z \to A,\ (t_p[Z] \parallel t_p[A]))$.

That is, there is neither redundant attributes in each CFD nor redundant CFDs in Σ_{mc}.

We only include nontrivial CFDs in Σ_{mc}. A CFD $R(X \to A,\ t_p)$ is *nontrivial* if either (a) $A \notin X$, or (b) $X = AZ$ but t_p is not of the form $(\eta_1, \overline{d_Z} \parallel \eta_2)$, where either $\eta_1 = \eta_2$, or η_1 is a constant and $\eta_2 =$ '_'.

It is known that without finite-domain attributes, implication of CFDs can be decided in quadratic time [FGJK08]. Further, there is an algorithm [FGJK08], referred to as MinCover, which computes Σ_{mc} in $O(|\Sigma|^3)$ time for any given set Σ of CFDs.

Propagation cover. For a view V defined on a schema $\mathcal{R}$ and a set Σ of source CFDs on $\mathcal{R}$, we denote by $\mathsf{CFD}_p(\Sigma, V)$ the set of *all* view CFDs propagated from Σ via V.

The *propagation cover problem* is to compute, given V and Σ, a cover Γ of $\mathsf{CFD}_p(\Sigma, V)$. We refer to Γ as a *propagation cover* of Σ via V, and as a *minimal* propagation cover if Γ is a minimal cover of $\mathsf{CFD}_p(\Sigma, V)$.

Challenges. The example below, taken from [FJT83], shows that the problem is al-

ready hard for FDs and simple P views.

Example 5.4.1: Consider a schema R with attributes A_i, B_i, C_i and D, and a set Σ of FDs on R consisting of $A_i \to C_i$, $B_i \to C_i$, and $C_1, \ldots, C_n \to D$, for each $i \in [1,n]$. Consider a view that projects an R relation onto their A_i, B_i and D attributes, dropping C_i's. Then any cover Σ_c of the set of view FDs propagated *necessarily contains* all FDs of the form $\eta_1, \ldots, \eta_n \to D$, where η_i is either A_i or B_i for $i \in [1,n]$. Obviously Σ_c contains at least 2^n FDs, whereas the size of the input, namely, Σ and the view, is $O(n)$. Indeed, to derive view FDs from $C_1, \ldots, C_n \to D$, one can substitute either A_i or B_i for each C_i, leading to the exponential blowup. □

In contrast, the dependency propagation problem is in PTIME in this setting (recall from Section 5.3). This shows the sharp distinction between the dependency propagation problem and the propagation cover problem.

While we are not aware of any previous methods for finding propagation covers for FDs via SPC views, there has been work on computing *embedded* FDs [Got87, SK86, Ull82], which is essentially to compute a propagation cover of FDs via projection views. Given a schema R, a set F of FDs on R and a set Y of attributes in R, it is to find a cover F_c of all FDs propagated from F via a projection view $\pi_Y(R)$. An algorithm for finding F_c is by first computing the closure F^+ of F, and then projecting F^+ onto Y, removing those FDs with attributes not in Y. This algorithm always takes $O(2^{|F|})$ time, for computing F^+. As observed in [Got87], this is the method covered in database texts [SK86, Ull82] for computing F_c. A more practical algorithm was proposed in [Got87], which we shall elaborate shortly.

Already hard for FDs and P views, the propagation cover problem is more intricate for CFDs and SPC views.

(a) While at most exponentially many FDs can be defined on a schema R, there are possibly infinitely many CFDs. Indeed, there are infinitely many CFDs of the form $R(A \to B, t_p)$ when $t_p[A]$ ranges over values from an infinite $\mathsf{dom}(A)$.

(b) While $AX \to A$ is a trivial FD and can be ignored, $\phi = R(AX \to A, t_p)$ may not be. Indeed, when t_p is $(\_, \overline{d_X} \| a)$, ϕ is meaningful: it asserts that for all tuples t such that $t[X] \asymp \overline{d_X}$, the A column has the same constant 'a'.

(c) While $X \to Y$ and $Y \to Z$ yield $X \to Z$ for FDs, the transitivity rule for CFDs has to take pattern tuples into account and is more involved than its FD counterpart [FGJK08].

(d) As we have seen in the previous section, selection and Cartesian product introduce interaction between domain constraints and CFDs, a complication of SPC views that we do not encounter when dealing with projection views.

5.4.2 Propagating CFDs via SPC Views

The exponential complexity of Example 5.4.1 is for the worst case and is only found in examples intentionally constructed. In practice, it is common to find a propagation cover of polynomial size, and thus it is an overkill to use algorithms that always take exponential time. In light of this one wants an algorithm for computing minimal propagation covers that behaves polynomially most of the time, whereas it necessarily takes exponential time only when all propagation covers are exponentially large for a given input.

We next propose such an algorithm, referred to as PropCFD_SPC, by extending the algorithm of [Got87] for computing a propagation cover of FDs via projection views. Given an SPC view V defined on a schema $\mathcal{R}$ and a set Σ of source CFDs on $\mathcal{R}$, PropCFD_SPC computes a minimal propagation cover Γ of Σ via V, without increasing the complexity of the algorithm of [Got87], although CFDs and SPC views are more involved than FDs and P views, respectively.

Before we present PropCFD_SPC, we give some basic results behind it. Let $\mathcal{R} = (S_1, \ldots, S_m)$ be the source schema. Recall from Section 5.2 that V defined on $\mathcal{R}$ is of the form:

$$\pi_Y(R_c \times E_s),\ E_s = \sigma_F(E_c),\ E_c = R_1 \times \ldots \times R_n$$

where R_c is a constant relation, R_j's are renamed relation atoms $\rho_j(S)$ for S in $\mathcal{R}$, Y is the set of projection attributes, and F is a conjunction of equality atoms.

Basic results. The constant relation R_c introduces no difficulties: for each $(A_i : a_i)$ in R_c, a CFD $\mathcal{R}_V(A_i \to A_i, (\_ \,\|\, a))$ is included in Γ, where $\mathcal{R}_V$ is the view schema derived from V and $\mathcal{R}$. Thus in the sequel we assume that $V = \pi_Y(E_s)$.

The reduction below allows us to focus on E_s instead of V. The proof is straightforward, by contradiction.

Proposition 5.4.1: *For any SPC view V of the form above, and any set Σ of source* CFD*s, $\Sigma \models_V \phi$ iff $\Sigma \models_{E_s} \phi$ when*

- $\phi = \mathcal{R}_V(A \to B, (x \,\|\, x))$, *denoting $A = B$;*
- $\phi = \mathcal{R}_V(A \to A, (\_ \,\|\, a))$, *denoting $A =$ 'a'; or*
- $\phi = \mathcal{R}_V(X \to A, t_p)$;

where $A \in Y$, $B \in Y$, and $X \subseteq Y$. □

We next illustrate how we handle the interaction between CFDs and operators $\times, \sigma$ and π in the view definition V.

Cartesian product. Observe that each R_j in E_c is $\rho_j(S)$, where S is in $\mathcal{R}$. All source

CFDs on S are propagated to the view, after their attributes are renamed via ρ_j.

Selection. The condition F in σ_F brings domain constraints into play, which can be expressed as CFDs.

Lemma 5.4.2: *(a) If $A = $ 'a' is in the selection condition F, then $\mathcal{R}_V(A \to A,\ (\_ \parallel a))$ is in $\mathsf{CFD}_p(\Sigma, V)$. (b) If $A = B$ is in F, then $\mathcal{R}_V(A \to B,\ (x \parallel x))$ is in $\mathsf{CFD}_p(\Sigma, V)$ for the special variable x.* □

That is, one can incorporate domain constraints $A =$ 'a' and $A = B$ enforced by the view V into CFDs. Here (a) asserts that the A column of the view must contain the same constant 'a', and (b) asserts that for each tuple t in the view, $t[A]$ and $t[B]$ must be identical, as required by the selection condition F in the view V (this is why we introduced CFDs of the form $\mathcal{R}_V(A \to B,\ (x \parallel x))$ in Section 5.2).

Lemma 5.4.3: *If $\mathcal{R}_V(A \to B,\ (x \parallel x))$ and $\mathcal{R}_V(BX \to G,\ t_p)$, then $\mathcal{R}_V(AX \to G,\ t'_p)$ is in $\mathsf{CFD}_p(\Sigma, V)$, where $t'_p[A] = t_p[B]$, $t'_p[X] = t_p[X]$ and $t'_p[G] = t_p[G]$.* □

That is, we can derive view CFDs by applying the domain constraint $A = B$: substituting A for B in a view CFD yields another view CFD. This also demonstrates how domain constraints interact with CFD propagation.

Let us use Σ_d to denote these CFDs as well as those in Σ expressing domain constraints. Based on Σ_d we can decide whether $A = B$ or $A =$ 'a' for attributes in Y (*i.e.,* R_V).

More specifically, we partition the attributes into a set EQ of equivalence classes, such that for any $\mathsf{eq} \in \mathsf{EQ}$, and for any attributes A, B in Y, (a) $A, B \in \mathsf{eq}$ iff $A = B$ can be derived from Σ_d; (b) if $A =$ 'a' can be derived from Σ_d and moreover, $A \in \mathsf{eq}$, then for any $B \in \mathsf{eq}$, $B =$ 'a'; we refer to the constant 'a' as the *key* of eq, denoted by $\mathsf{key}(\mathsf{eq})$. If a constant is not available, we let $\mathsf{key}(\mathsf{eq})$ be '_'.

The use of EQ helps us decide whether or not V and Σ always yield empty views (Section 5.3), which happens if there exists some $\mathsf{eq} \in \mathsf{EQ}$ such that $\mathsf{key}(\mathsf{eq})$ is not well-defined, *i.e.,* when two distinct constants are associated with eq.

It is easy to develop a procedure to compute EQ, referred to as $\mathsf{ComputeEQ}$, which takes Σ and V as input, and returns EQ as output, along with $\mathsf{key}(\mathsf{eq})$ for each $\mathsf{eq} \in \mathsf{EQ}$. If $\mathsf{key}(\mathsf{eq})$ is not well-defined for some eq, it returns a special symbol '$\perp$', indicating the inconsistency in V and Σ.

Projection. To remedy the limitations of closure-based methods for computing propagation covers of FDs via P views, a practical algorithm was proposed in [Got87] based on the idea of *Reduction by Resolution* (RBR). We extend RBR and the algorithm

of [Got87] to handle CFDs and projection.

To illustrate RBR, we first define a partial order $\leq$ on constants and '_': $\eta_1 \leq \eta_2$ if either η_1 and η_2 are the same constant 'a', or η_2 = '_'.

Given CFDs $\phi_1 = R(X \to A, t_p)$ and $\phi_2 = R(AZ \to B, t'_p)$, if $t_p[A] \leq t'_p[A]$ and for each $C \in X \cap Z$, $t_p[C] \asymp t'_p[C]$, then we can derive $\phi = R(XZ \to B, s_p)$ based on CFD implication [FGJK08]. Here $s_p = (t_p[X] \oplus t'_p[Z] \parallel t'_p[B])$, and $t_p[X] \oplus t'_p[Z]$ is defined as follows:

- for each $C \in X - Z$, $s_p[C] = t_p[C]$;
- for each $C \in Z - X$, $s_p[C] = t'_p[C]$;
- for each $C \in X \cap Z$, $s_p[C] = \mathsf{min}(t_p[C], t'_p[C])$, *i.e.,* the smaller one of $t_p[C]$ and $t'_p[C]$ if either $t_p[C] \leq t'_p[C]$ or $t'_p[C] \leq t_p[C]$; it is undefined otherwise.

Following [Got87], we refer to ϕ as an A-resolvent of ϕ_1 and ϕ_2.

Example 5.4.2: Consider CFDs $\phi_1 = R([A_1, A_2] \to A,\ t_1)$ and $\phi_2 = R([A, A_2, B_1] \to B,\ t_2)$, where $t_1 = (\_, c \parallel a)$ and $t_2 = (\_, c, b \parallel \_)$. Then $\phi = R([A_1, A_2, B_1] \to B,\ t_p)$ is an A-resolvent of ϕ_1 and ϕ_2, where $t_p = (\_, c, b \parallel \_)$. □

Following [Got87], we define the following. Given $\pi_Y(R)$ and a set Σ of CFDs on R, let U be the set of attributes in R.

- For $A \in (U - Y)$, let $\mathsf{Res}(\Sigma, A)$ denote the set of all nontrivial A-resolvents. Intuitively, it shortcuts all CFDs involving A.
- Denote by $\mathsf{Drop}(\Sigma, A)$ the set $\mathsf{Res}(\Sigma, A) \cup \Sigma[U - \{A\}]$, where $\Sigma[Z]$ denotes the subset of Σ by including only CFDs with attributes in Z.
- Define $\mathsf{RBR}(\Sigma, A) = \mathsf{Drop}(\Sigma, A)$ and inductively, $\mathsf{RBR}(\Sigma, ZA) = \mathsf{Drop}(\mathsf{Drop}(\Sigma, A), Z)$.

Then we have the following result, in which F^+ denotes the closure of F, *i.e.,* the set of all CFDs implied by F.

Proposition 5.4.4: *For a view $\pi_Y(R)$ and a set Σ of CFDs on R, (a) for each $A \in (U - Y)$, $\mathsf{Drop}(\Sigma, A)^+ = \Sigma^+[U - \{A\}]$; (b) $\mathsf{RBR}(\Sigma, U - Y)$ is a propagation cover of Σ via $\pi_Y(R)$, where U is the set of attributes in R.* □

The result is first established in [Got87] for FDs. The proof of Proposition 5.4.4 is an extension of its counterpart in [Got87].

This yields a procedure for computing a propagation cover of Σ via $\pi_Y(R)$, also denoted by RBR. The idea is to repeatedly "drop" attributes in $U - Y$, shortcutting all CFDs that involve attributes in $U - Y$. The procedure takes as input Σ and $\pi_Y(R)$, and returns $\mathsf{RBR}(\Sigma, U - Y)$ as the output.

Input: Source CFDs Σ, and SPC view $V = \pi_Y(E_s)$, $E_s = \sigma_F(E_c)$.
Output: A minimal propagation cover of Σ via V.
1. $\Sigma_V := \emptyset$; $\Sigma := \mathsf{MinCover}(\Sigma)$;
2. $\mathsf{EQ} := \mathsf{ComputeEQ}(E_s, \Sigma)$; /* handling σ_F */
3. **if** $\mathsf{EQ} = \bot$ /* inconsistent */
4. **then return** $\{R_V(A \to A, (\_ \,\|\, a)), R_V(A \to A, (\_ \,\|\, b))\}$;
   /* for some $A \in Y$, distinct $a, b \in \mathsf{dom}(A)$ */
5. **for each** $R_j = \rho_j(S)$ in E_c **do**
6. $\quad \Sigma_V := \Sigma_V \cup \rho_j(\Sigma)$; /* handling product '$\times$' */
7. **for each** $\mathsf{eq} \in \mathsf{EQ}$ **do** /* applying domain constraints */
8. $\quad$ pick an attribute $\mathsf{rep}(\mathsf{eq})$ in eq
   $\quad\quad$ such that $\mathsf{rep}(\mathsf{eq}) \in Y$ if $\mathsf{eq} \cap Y$ is not empty;
9. $\quad$ substitute $\mathsf{rep}(\mathsf{eq})$ for each $A \in \mathsf{eq}$, in Σ_V;
10. $\quad \mathsf{eq} := \mathsf{eq} \cap Y$; /* keep only those attributes in Y */
11. $\Sigma_c := \mathsf{RBR}(\Sigma_V, \mathsf{attr}(E_s) - Y)$; /* handling π_Y */
    /* $\mathsf{attr}(E_s)$ is the set of attributes in E_s */
12. $\Sigma_d := \mathsf{EQ2CFD}(\mathsf{EQ})$; /* put domain constraints as CFDs */
13. **return** $\mathsf{MinCover}(\Sigma_c \cup \Sigma_d)$;

Figure 5.3: Algorithm PropCFD_SPC

We shall also need the following lemma.

Lemma 5.4.5: *If for any source instance D where $D \models \Sigma$, $V(D)$ is empty, then $\mathcal{R}_V(A \to A,\ (\_ \,\|\, a))$ and $\mathcal{R}_V(A \to A,\ (\_ \,\|\, b))$ are in $\mathsf{CFD}_p(\Sigma, V)$, for any attribute A in $\mathcal{R}_V$ and any distinct values $a, b \in \mathsf{dom}(A)$.* □

This essentially assures that the view is always empty (recall the emptiness problem from Section 5.3.4): no tuple t in the view can possibly satisfy the CFDs, which require $t[A]$ to take distinct a and b as its value. As a result any CFD on the view can be derived from these "inconsistent" CFDs.

5.4.3 An Algorithm for Computing Minimal Covers

We now present algorithm PropCFD_SPC, shown in Fig. 5.3.

The algorithm first processes selection σ_F (line 2), extracting equivalence classes EQ via procedure ComputeEQ described earlier (not shown). If inconsistency is discovered, it returns a pair of "conflicting" view CFDs that assure the view to be always

empty (lines 3-4), by Lemma 5.4.5. It then processes the Cartesian product, and gets a set Σ_V of CFDs via renaming as described above (lines 5-6). It applies the domain constraints of EQ to Σ_V (line 9), by designating an attribute rep(eq) for each equivalence class eq in EQ (line 8), which is used uniformly wherever attributes in eq appear in CFDs of Σ_V. It also removes attributes in eq that are not in the projection list Y (line 10), since these attributes do not contribute to view CFDs. Next, it handles the projection π_Y, by invoking procedure RBR (line 11), and converts domain constraints of EQ to CFDs via procedure EQ2CFD (line 12). Finally, it returns a minimal cover of the results returned by these procedures, by invoking procedure MinCover of [FGJK08]. This yields a minimal propagation cover of V via Σ (line 13).

Procedure RBR of Fig. 5.4 implements the RBR method: for each $A \in U - Y$, it computes an A-resolvent (lines 4-8) and Drop(Σ, A) (lines 4-11). Only nontrivial CFDs are included (line 8). By dropping all attributes in $U - Y$, it obtains RBR$(\Sigma, U - Y)$, a cover of Σ and π_Y by Proposition 5.4.4.

Procedure EQ2CFD of Fig. 5.5 converts domain constraints enforced by EQ to equivalent CFDs (lines 2-8), by Lemma 5.4.2. For each eq in EQ, it leverages the constant key(eq) whenever it is available (lines 4-5). When it is not available, it uses the special variable x in the CFDs (lines 6-8).

Example 5.4.3: Consider sources $R_1(B'_1, B_2)$, $R_2(A_1, A_2, A)$, $R_3(A', A'_2, B_1, B)$, and view $V = \pi_Y(\sigma_F(R_1 \times R_2 \times R_3))$, where $Y = \{B_1, B_2, B'_1, A_1, A_2, B\}$, and F is ($B_1 = B'_1$ **and** $A = A'$ **and** $A_2 = A'_2$). Consider Σ consisting of $\psi_1 = R_2([A_1, A_2] \to A,\ t_1)$ and $\psi_2 = R_3([A', A_2, B_1] \to B,\ t_2)$, for t_1, t_2 given in Example 5.4.2.

Applying algorithm PropCFD_SPC to Σ and V, after step 10, EQ consists of $\{\{B_1, B'_1\}, \{B_2\}, \{A_1\}, \{A_2\}, \{B\}\}$, and Σ_V consists of ϕ_1, ϕ_2 of Example 5.4.2. As also given there, procedure RBR returns ϕ of Example 5.4.2. Procedure EQ2CFD returns $\phi' = R(B_1 \to B'_1, (x \parallel x))$, where R is the view schema with attributes in Y. Then the cover returned by the algorithm consists of ϕ and ϕ'. □

Analysis. For the correctness of the algorithm, we show that for each ϕ in CFD$_p(\Sigma, V)$, $\Gamma \models \phi$, and vice versa, where Γ is the output of the algorithm. For both directions the proof leverages the lemmas and propositions given above.

For the complexity, let $V = \pi_Y(\sigma_F(E_c))$. Then $|Y| \leq |E_c|$ and $|F| \leq (|E_c|^2 + |E_c|)$. We have the following. (a) Procedure ComputeEQ takes $O(|E_c|^4 * |\Sigma|)$ time. (b) EQ2CFD is in $O(|Y|^3)$ time. (c) Procedure RBR has the same complexity as its counterpart in [Got87]: $O(|E_c|^2 * a^3)$, where a is an upper bound for the cardinality of Γ during the execution of RBR [Got87]. (d) The rest of the computation takes at

Input: An attribute set $X = U - Y$, and a set Σ of CFDs on U.

Output: A cover Γ of $\Sigma^+[Y]$.

1. **let** $\Gamma := \Sigma$;
2. **while** X is not empty **do** $\{$
3. pick $A \in X$; X :=$X - \{A\}$; $C := \emptyset$;
4. **for** each CFD $(W \to A, t_1) \in \Gamma$ **do**
5. **for** each CFD $(AZ \to B, t_2) \in \Gamma$ **do**
6. **if** $t_1[A] \preceq t_2[A]$ **and** $t_1[W] \oplus t_2[Z]$ is well defined
7. **then** $\phi := \mathcal{R}_V(WZ \to B,\ (t_1[W] \oplus t_2[Z] \parallel t_2[B]))$;
8. **if** ϕ is not trivial **then** C:=$C \cup \{\phi\}$;
9. **for** each CFD $\varphi \in \Gamma$ **do**
10. **if** A occurs in φ **then** Γ :=$\Gamma - \{\varphi\}$;
11. Γ :=$\Gamma \cup C$;$\}$
12. **return** Γ.

Figure 5.4: Procedure RBR

Input: A set EQ of attribute equivalence classes.

Output: A set Σ_d of CFDs expressing EQ.

1. $\Sigma_d := \emptyset$;
2. **for** each attribute equivalence class eq $\in$ EQ **do**
3. **if** key(eq) is a constant **then**
4. **for each** attribute $A \in$ eq
5. Σ_d :=$\Sigma_d \cup \{R_V(A \to A, (\_ \parallel \text{key(eq)}))\}$;
6. **if** key(eq)='_' **then**
7. **for each** $A, B \in$ eq **do**
8. Σ_d :=$\Sigma_d \cup \{R_V(A \to B, (x \parallel x))\}$;
9. **return** Σ_d;

Figure 5.5: Algorithm EQ2CFD

most $O(|\Sigma|^3 + a^3 + |E_c|^2)$ time. Since a is no less than $|E_c| * |\Sigma|$, RBR takes at least $O(|E_c|^5 * |\Sigma|^3)$ time. Putting these together, clearly the cost of RBR dominates. That is, the complexity of PropCFD_SPC is the same as the bound on the algorithm of [Got87]. Note that both Σ and V are defined at the schema level (it has nothing to do with the instances of source databases), and are often small in practice.

A number of practical cases are identified by [Got87], where RBR is in polynomial time. In all these cases, PropCFD_SPC also behaves polynomially.

We use minimal cover as an optimization technique. First, Σ is "simplified" by invoking MinCover(Σ) (line 1 of Fig. 5.3), removing redundant source CFDs. Second (not shown), in procedure RBR, to reduce the size of intermediate Γ during the computation, one can change line 11 of Fig. 5.4 to "Γ := MinCover($\Gamma \cup C$)". In our implementation, we partition Γ into $\Gamma_1, \ldots, \Gamma_k$, each of a fixed size k_0, and invoke MinCover(Γ_i). This removes redundant CFDs from Γ, to an extent, without increasing the worst-case complexity since it takes $O(|\Gamma| * k_0^2)$ time to conduct.

As another optimization technique, one may simplify or better still, minimize input SPC views. This works, but only to an extent: the minimization problem for SPC queries is intractable (see, *e.g.,* [AHV95] for a detailed discussion).

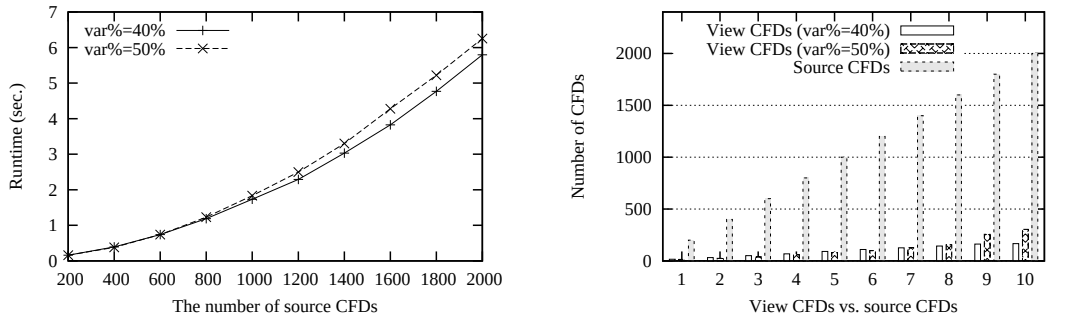
5.5 Experimental Study

In this section, we present an experimental study of algorithm PropCFD_SPC for computing minimal propagation covers of CFDs via an SPC view. We investigate the effects of the number of source CFDs and the complexity of SPC views on the performance of PropCFD_SPC. We also evaluate the impacts of these factors on the cardinality of the minimal propagation covers computed by PropCFD_SPC.

Experimental Setting. We designed two generators to produce CFDs and SPC views, on which our experiments are based. We considered source relational schemas $\mathcal{R}$ consisting of at least 10 relations, each with 10 to 20 attributes.

(a) CFD generator. Given a relational schema $\mathcal{R}$ and two natural numbers m and n, the CFD generator randomly produces a set Σ consisting of m source CFDs defined on $\mathcal{R}$, such that the average number of CFDs on each relation in $\mathcal{R}$ is n. The generator also takes another two parameters LHS and var% as input: LHS is the maximum number of attributes in each CFD generated, and var% is the percentage of the attributes which are filled with '_' in the pattern tuple, while the rest of the attributes draw random values from their corresponding domains. Note that LHS and var% indicate how complex the CFDs are. The experiments were conducted on various Σ's ranging from 200 to 2000 CFDs, with LHS from 3 to 9 and var% from 40% to 50%.

(b) SPC view generator. Given a source schema $\mathcal{R}$ and three numbers $|Y|, |F|$ and $|E_c|$, the view generator randomly produces an SPC view $\pi_Y(\sigma_F(E_c))$ defined on $\mathcal{R}$ such that the set Y consists of $|Y|$ projection attributes, the selection condition F is a conjunction of $|F|$ domain constraints of the form $A = B$ and $A =$ 'a', and E_c is the Cartesian product of $|E_c|$ relations. Here each constant a is randomly picked from



(a) Scalability *w.r.t.* the number of source CFDs (b) The cardinality of minimal propagation cover

Figure 5.6: Varying source CFDs

a fixed range $[1, 100000]$ such that the domain constraints may interact with each other. The complexity of an SPC view is determined by Y, F and E_c. In the experiments we considered $|Y|$ ranging from 5 to 50, $|F|$ from 1 to 10, and $|E_c|$ from 2 to 11.

The algorithm was implemented in Java. The experiments were run on a machine with a 3.00GHz Intel(R) Pentium(R) D processor and 1GB of memory. For each experiment, we randomly generated 10 sets of source CFDs (resp. SPC views) with fixed parameters, and ran the experiments 5 times on each of the datasets. The average is reported here.

Experimental results. We conducted two sets of experiments: one focused on the scalability of the algorithm and the cardinality of minimal propagation covers *w.r.t.* various source CFDs, while the other evaluated these *w.r.t.* the complexity of SPC views.

Varying CFDs on the Source. To evaluate the impact of source CFDs on the performance of PropCFD_SPC, we fixed $|Y| = 25$, $|F| = 10$, $|E_c| = 4$, and varied the set Σ of source CFDs. More specifically, we considered Σ with $|\Sigma|$ ranging from 200 to 2000, *w.r.t.* var% = 40% and var% = 50%, while the number of attributes in each CFD ranged from 3 to 9 (LHS = 9). The running time and the cardinality of minimal propagation covers are reported in Fig. 5.6.

Figure 5.6(a) shows that PropCFD_SPC scales well with $|\Sigma|$ and is rather efficient: it took less than 7 seconds for $|\Sigma| = 2000$. Further, the algorithm is not very sensitive to (var%, LHS): the results for various (var%, LHS) are quite consistent. As we will see, however, when Y also varies, the algorithm is sensitive to Y and (var%, LHS) taken together.

Figure 5.6(b) tells us that the more source CFDs are given, the larger the minimal propagation cover found by the algorithm is, as expected. It is interesting to note that the cardinality of the minimal propagation cover is even smaller than the number $|\Sigma|$ of the source CFDs. This confirms the observation of [Got87]: (minimal) propagation

covers are typically much smaller than an exponential of the input size, and thus there is no need to pay the price of the exponential complexity of the closure-based methods to compute covers.

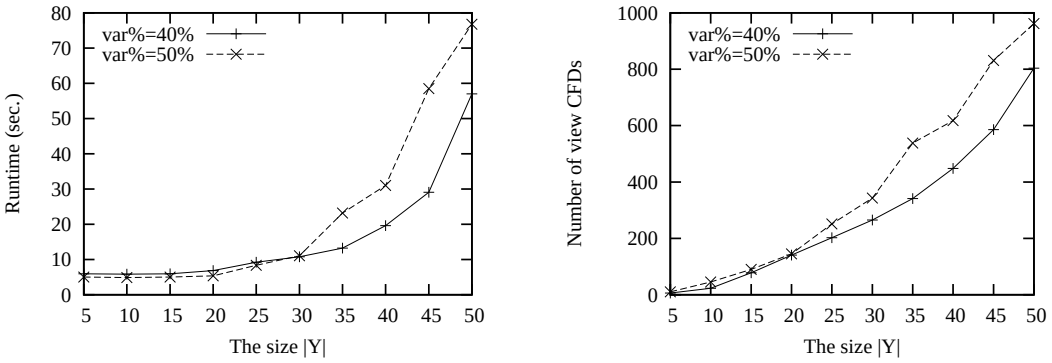
Varying the complexity of SPC views. In the second set of experiments, we evaluated the performance of algorithm PropCFD_SPC *w.r.t.* each of the following parameters of SPC views $\pi_Y(\sigma_F(E_c))$: the set Y of projection attributes, the selection condition F, and the Cartesian product E_c. In each of the experiments, we considered sets Σ of source CFDs with $|\Sigma| = 2000$, *w.r.t.* var% = 40% and var% = 50%, while the number of attributes in each CFD ranged from 3 to 9.

(a) We first evaluated the scalability of PropCFD_SPC with Y: fixing $|F| = 10$ and $|E_c| = 4$, we varied $|Y|$ from 5 to 50. The results are reported in Fig. 5.7(a), which shows that the algorithm is very sensitive to $|Y|$. On one hand, the larger $|Y|$, the smaller $|U - Y|$, where U is the total number of attributes in E_c (which is determined by $|E_c|$ and the arities of the relations in the source schema $\mathcal{R}$). Note that the outer loop of procedure RBR is dominated by $|U - Y|$: the larger $|U - Y|$ is, the more iterations RBR performs. On the other hand, the larger $|Y|$ is, the more source CFDs are propagated to the view, which has bigger impact on the performance of RBR. As a combination of the above two factors, the running time of PropCFD_SPC does not increase much when $|Y|$ ranges from 5 to 30. However, the running time increases rather rapidly when $|Y|$ is beyond 30. The good news is that even when $|Y|$ is 50 (with $|\Sigma| = 2000$), the algorithm took no more than 80 seconds.

Further, Figure 5.7(a) shows that when $|Y| < 30$, different settings of var% do not make much difference. However, when $|Y| \geq 30$, their impact on the performance of PropCFD_SPC becomes more obvious. This is because constants may block the transitivity (and thus propagation) of CFDs in procedure RBR (lines 4-8 of Fig. 5.4); thus more CFDs are propagated to the view when there are less constants (larger var%). When $|Y|$ is small, the impact is not obvious since only a small number of CFDs are propagated to the view. Note that Figures 5.7(a) and 5.6(a) are consistent: in the experiments for Fig. 5.6(a), $|Y|$ was fixed to be 25.

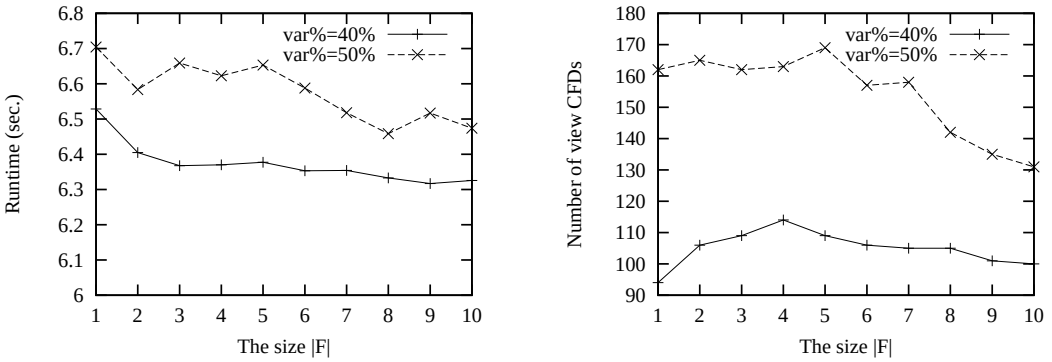
Figure 5.7(b) shows that when $|Y|$ or var% gets larger, more source CFDs are propagated to the view, as expected. Again it confirms that the minimal covers found are smaller than the source CFDs even when $|Y|$ reached 50.

(b) We next evaluated the scalability of PropCFD_SPC with the selection condition F of SPC views. We fixed $|Y| = 25$ and $|E_c| = 4$, and varied $|F|$ from 1 to 10. The results are reported in Fig. 5.8. Figure 5.8(a) shows that when $|F|$ increases, the running time



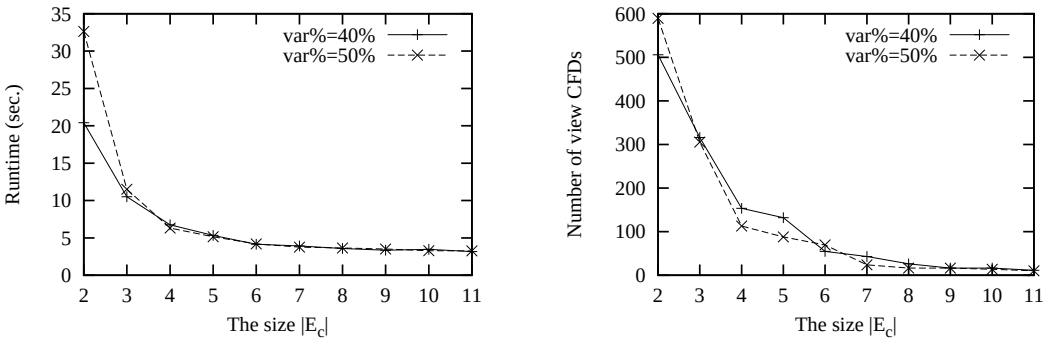
(a) Scalability *w.r.t.* the size $|Y|$ (b) The number of CFDs propagated

Figure 5.7: Varying the number of projection attributes in Y in SPC views



(a) Scalability *w.r.t.* the size $|F|$ (b) The number of CFDs propagated

Figure 5.8: Varying the selection condition F in SPC views



(a) Scalability *w.r.t.* the size $|E_c|$ (b) The number of CFDs propagated

Figure 5.9: Varying the number of relations in the Cartesian product E_c in SPC views

decreases. This is because F introduces domain constraints, which interact with source CFDs, and may either make those CFDs trivial, or combine multiple CFDs into one (see line 9 of Fig. 5.3). As a result, the larger $|F|$ is, the smaller the set Σ_V is, which is passed to procedure RBR (lines 10-11 of Fig. 5.3). This leads to the decrease in the running time of RBR, and in turn, the decrease in the running time of PropCFD_SPC. This is rather consistent for the two settings of var%.

Figure 5.8(b) shows the cardinality of minimal propagation covers *w.r.t.* various $|F|$. The cardinality went up and then down. This is because when $|F|$ increases, to an extent (less than 4 for var% = 40% or 5 for var% = 50%, Fig. 5.8(b)), more domain constraints are propagated to the view. However, when $|F|$ gets larger, the interaction between domain constraints and CFDs takes a lager toll and as a result, fewer source CFDs are propagated to the view, for the reason given above. This leads to the decrease in the cardinality of the minimal covers when $|F|$ is large. Figure 5.8(b) also confirms that when var% gets larger, more source CFDs are propagated to the view, which is consistent with Fig. 5.7(b).

(c) Finally, we evaluated the impact of E_c on the performance of PropCFD_SPC. Fixing $|F| = 10$ and $|Y| = 25$, we varied $|E_c|$ from 2 to 11. The results are reported in Fig. 5.9.

Figure 5.9(a) shows that when $|E_c|$ gets larger, the algorithm takes less time. This is because when Y, F and Σ are fixed, increasing $|E_c|$ leads to more CFDs to be dropped from the minimal propagation cover, for involving attributes not in Y. Further, when $|E_c|$ gets larger, *i.e.,* when the total number of attributes involved gets larger, F is less effective in identifying attributes in Y and those not in F. As an example, consider two views defined on the same source: $V_1 = \pi_{AB}(\sigma_{C=D}(R_1(ABCD)))$, and $V_2 = \pi_{AE}(\sigma_{C=H}(R_1(ABCD) \times R_2(EGHL)))$, while the set Σ consists of source CFDs $R_1(A \to B,\ (\_ \parallel \_))$ and $R_2(E \to L,\ (\_ \parallel \_))$. Let V_1, V_2 also denote the view schema of V_1, V_2, respectively. Then a minimal cover of the CFDs propagated via V_1 consists of $V_1(A \to B,\ (\_ \parallel \_))$ whereas no nontrivial CFDs are propagated to the view via V_2.

Figure 5.9(a) also shows that when $|E_c| \geq 6$, the algorithm is insensitive to E_c, because most of the sources CFDs are dropped, *i.e.,* not propagated to the view (as $|\Sigma|$ is fixed).

For the same reason, when $|E_c|$ gets larger, the minimal propagation covers get smaller, as shown in Fig. 5.9(b). However, different from Fig. 5.7(b) and Fig. 5.8(b), the number of CFDs propagated in this case is insensitive to different settings of var%. This is because the effect of $|E_c|$ outweighs that of var% in this experiment.

Summary. We have presented several results from our experimental study of algorithm PropCFD_SPC. From the results we find the following. First, the algorithm is quite efficient; for example, it took less than 80 seconds when $|\Sigma| = 2000$, $|Y| = 50$, $|F| = 10$ and $|E_c| = 4$. Second, it scales well with the input set Σ of source CFDs, the selection condition F and the number of relations involved in Cartesian product E_c in the input SPC views. In contrast, the cost increases rapidly when the set Y of projection attributes gets large. Nonetheless, as remarked above, the cost is not unbearable:

when $|Y| = 50$ the algorithm still performed reasonably well. Third, we note that minimal propagation covers found by the algorithm are typically small, often smaller than the input set Σ of source CFDs. Finally, while the algorithm is quite sensitive to Y, it is less sensitive to the other complexity factors F and E_c of SPC views. Further, the complexity factors (var%, LHS) of source CFDs do not have significant impact on the performance of the algorithm when the size of Y is less than 30.

5.6 Related Work

To our knowledge, no previous work has considered (a) CFD propagation, (b) dependency propagation in the general setting, FDs or CFDs, and (c) methods for computing minimal propagation covers for SPC views, even for FDs.

Closest to the study of the CFD propagation problem are [Klu80, KP82], which are the first to investigate dependency propagation. The undecidability result for FD propagation via RA views is shown in [Klu80]. An extension of chase is given in [KP82], for FDs and beyond, based on which the PTIME complexity of FD propagation via SPCU views is derived by [AHV95]. Our proofs of the results in Sections 5.3 further extend the chase of [KP82], to accommodate CFDs. As remarked earlier, [Klu80, KP82, AHV95] assume the absence of finite-domain attributes. This work extends [Klu80, KP82, AHV95] by providing complexity bounds for FD propagation in the general setting, and for CFD propagation in the infinite-domain setting and in the general setting.

As remarked in Section 5.4, prior work on propagation covers has focused on FDs and projection views, in the absence of finite-domain attributes [Got87, SK86, Ull82]. The first algorithm for computing covers *without* computing closures is proposed by [Got87], based on the RBR method. Our algorithm of Section 5.4 is inspired by [Got87], and also uses RBR to handle the interaction between CFDs and the projection operator. This work is the first that deals with selection, Cartesian product and projection operators for computing propagation covers.

Dependency propagation has also been studied for other models [DFHQ03, HL04, PT99]. Propagation from XML keys to relational FDs is studied in [DFHQ03], and composition of views and a powerful set of constraints is investigated for object-oriented databases in [PT99]. The complexity bounds and techniques developed there do not apply to CFD propagation. An extension of FDs, and their interaction with schema transformation operators (*e.g.,* folding, unfolding) are considered in [HL04]. Those ex-

tended FDs and transformation operators are very different from CFDs and SPC views, respectively.

There has also been a host of work on satisfaction families of dependencies, *e.g.,* [Fag82, GZ82, Hul85]. The focus is on the closure problem for dependencies under views. It is to decide, given a set Σ of dependencies and a view V, whether the set Γ of dependencies propagated from Σ via V characterizes the views, *i.e.,* whether the set of databases that satisfy Γ is precisely the set of views $V(D)$ where $D \models \Sigma$. It is shown [GZ82] that FDs are not closed under projection views. This work provides another example for that FDs are not closed under SPCU views: as shown in Section 5.1, FDs may be propagated to CFDs but not to standard FDs. While CFDs are not closed under SPC views either, the analysis of CFD propagation allows us to preserve certain important semantics of the source data that traditional FDs fail to capture.

The notion of CFDs is proposed in [FGJK08], which also studies the implication and consistency problems for CFDs. The propagation problem is *not* considered in [FGJK08].

A variety of extensions of classical dependencies have been proposed, for specifying constraint databases [BCW99, BP83, Mah97, MS96]. Constraints of [BP83, Mah97] cannot express CFDs. More expressive are constraint-generating dependencies (CGDs) of [BCW99] and constrained tuple-generating dependencies (CTGDs) of [MS96], both subsuming CFDs. A CGD is of the form $\forall\bar{x}(R_1(\bar{x}) \wedge \ldots \wedge R_k(\bar{x}) \wedge\xi(\bar{x}) \rightarrow \xi'(\bar{x}))$, where R_i's are relation atoms, and ξ,ξ' are arbitrary constraints that may carry constants. A CTGD is of the form $\forall\bar{x}(R_1(\bar{x}) \wedge \ldots \wedge R_k(\bar{x}) \wedge \xi \rightarrow \exists\bar{y}(R'_1(\bar{x},\bar{y}) \wedge \ldots \wedge R'_s(\bar{x},\bar{y}) \wedge \xi'(\bar{x},\bar{y})))$, subsuming both CINDs and TGDs. The increased expressive power of CGDs and CTGDs comes at the price of a higher complexity for reasoning about these dependencies. For example, the implication problem for CGDs is already coNP-complete even when all involved attributes have an infinite domain, while in contrast, its CFD counterpart is in quadratic time. Detailed discussions about the connections between CFDs and these extensions can be found in [FGJK08]. *No previous work* has studied propagation analysis of these extensions via views.

There has also been recent work on specifying dependencies for XML in terms of description logics [TW06, TW05]. These dependencies also allow constants (concepts). However, the implication problem for such dependencies is undecidable, and as a result, their propagation problem is also undecidable even for views defined as identity mappings.

Chapter 6

Dynamic Constraints for Record Matching

This chapter investigates constraints for matching records from *unreliable* data sources. (a) We introduce a class of *matching dependencies* (MDs) for specifying the semantics of unreliable data. As opposed to static constraints for schema design, MDs are developed for record matching, and are defined in terms of *similarity metrics* and a *dynamic semantics*. (b) We identify a special case of MDs, referred to as *relative candidate keys* (RCKs), to determine what attributes to compare and how to compare them when matching records across possibly different relations. (c) We propose a mechanism for inferring MDs, a departure from traditional implication analysis, such that when we cannot match records by comparing attributes that contain errors, we may still find matches by using other, more reliable attributes. (d) We develop a sound and complete system for inferring MDs. (e) We provide a quadratic-time algorithm for inferring MDs, and an effective algorithm for deducing a set of quality RCKs from MDs. (f) We experimentally verify that the algorithms help matching tools efficiently identify keys at compile time for matching, blocking or windowing, and in addition, that the MD-based techniques improve the quality and efficiency of various record matching methods.

6.1 Introduction

Record matching is the problem for identifying tuples in one or more relations that refer to the same real-world entity. This problem is also known as record linkage, merge-purge, data deduplication, duplicate detection and object identification. The need for record matching is evident. In data integration it is necessary to collate in-

	c#	SSN	FN	LN	addr	tel	email	gender	type
t_1:	111	079172485	Mark	Clifford	10 Oak Street, MH, NJ 07974	908-1111111	mc@gm.com	M	master
t_2:	222	191843658	David	Smith	620 Elm Street, MH, NJ 07976	908-2222222	dsmith@hm.com	M	visa

(a) Example credit relation I_c

	c#	FN	LN	post	phn	email	gender	item	price
t_3:	111	Marx	Clifford	10 Oak Street, MH, NJ 07974	908	mc	null	iPod	169.99
t_4:	111	Marx	Clifford	NJ	908-1111111	mc	null	book	19.99
t_5:	111	M.	Clivord	10 Oak Street, MH, NJ 07974	1111111	mc@gm.com	null	PSP	269.99
t_6:	111	M.	Clivord	NJ	908-1111111	mc@gm.com	null	CD	14.99

(b) Example billing relation I_b

Figure 6.1: Example credit and billing relations

formation about an object from multiple data sources [LSPR96]. In data cleaning it is critical to eliminate duplicate records [BS06]. In master data management one often needs to identify links between input tuples and master data [Los09]. The need is also highlighted by payment card fraud, which cost $4.84 billion worldwide in 2006 [Fra]. In fraud detection it is a routine process to cross-check whether a card user is the legitimate card holder.

Record matching is a longstanding issue that has been studied for decades. A variety of approaches have been proposed for record matching: probabilistic [FS69, Jar89, Yan07, Win02], learning-based [CR02, SB02, VEH02], distance-based [GKMS04], and rule-based [ACG02, HS95, LSPR96] (see [EIV07] for a recent survey).

No matter what approach to use, one often needs to decide what attributes to compare and how to compare them. Real life data is typically dirty (*e.g.,* a person's name may appear as "Mark Clifford" and "Marx Clifford"), and may not have a uniform representation for the same object in different data sources. To cope with these it is often necessary to hinge on the semantics of the data. Indeed, domain knowledge about the data may tell us what attributes to compare. Moreover, by analyzing the semantics of the data we can deduce alternative attributes to inspect such that when matching cannot be done by comparing attributes that contain errors, we may still find matches by using other, more reliable attributes. This is illustrated by the following example.

Example 6.1.1: Consider two data sources specified by the following relation schemas:

credit (c#, SSN, FN, LN, addr, tel, email, gender, type),
billing (c#, FN, LN, post, phn, email, gender, item, price).

Here a credit tuple specifies a credit card (with number c# and type) issued to a card holder who is identified by SSN, FN (first name), LN (last name), addr (address), tel (phone), email and gender. A billing tuple indicates that the price of a purchased item

is paid by a credit card of number c#, used by a person specified in terms of name (FN, LN), gender, postal address (post), phone (phn) and email. An example instance (I_c, I_b) of (credit, billing) is shown in Fig. 6.1.

For payment fraud detection, one needs to check whether for any tuple t in I_c and any tuple t' in I_b, if $t[\mathsf{c\#}] = t'[\mathsf{c\#}]$, then $t[Y_c]$ and $t'[Y_b]$ refer to the same person, where Y_c and Y_b are two attribute lists:

$$Y_c = [\mathrm{FN}, \mathrm{LN}, \mathsf{addr}, \mathsf{tel}, \mathsf{gender}], \text{ and } Y_b = [\mathrm{FN}, \mathrm{LN}, \mathsf{post}, \mathsf{phn}, \mathsf{gender}].$$

Due to errors in the data sources one may not be able to match $t[Y_c]$ and $t'[Y_b]$ via pairwise comparison of their attributes. In the instance of Fig. 6.1, for example, billing tuples t_3–t_6 and credit tuple t_1 actually refer to the same card holder. However, *no match* can be found when we check whether the Y_b attributes of t_3–t_6 and the Y_c attributes of t_1 are identical.

Domain knowledge about the data suggests that we only need to compare LN, FN and address when matching $t[Y_c]$ and $t'[Y_b]$ [HS95]: if a credit tuple t and a billing tuple t' have the same address and last name, and if their first names are similar (although they may not be identical), then the two tuples refer to the same person. That is, LN, FN and address are a "key" for matching $t[Y_c]$ and $t'[Y_b]$:

- If $t[\mathrm{LN}, \mathsf{addr}] = t'[\mathrm{LN}, \mathsf{post}]$ and if $t[\mathrm{FN}]$ and $t'[\mathrm{FN}]$ are *similar w.r.t.* a similarity function $\approx_d$, then $t[Y_c]$ and $t'[Y_b]$ are a match.

Such a *matching key* tells us what attributes to compare and how to compare them in order to match $t[Y_c]$ and $t'[Y_b]$. By comparing only the attributes in the key we can now match t_1 and t_3, although their FN, tel, email and gender attributes are not identical.

A closer examination of the data semantics further suggests the following: for any credit tuple t and billing tuple t',

- if $t[\mathsf{email}] = t'[\mathsf{email}]$, then we can identify $t[\mathrm{LN}, \mathrm{FN}]$ and $t'[\mathrm{LN}, \mathrm{FN}]$, *i.e.,* they should be equalized via updates;
- if $t[\mathsf{tel}] = t'[\mathsf{phn}]$, then we can identify $t[\mathsf{addr}]$ and $t'[\mathsf{post}]$.

None of these makes a key for matching $t[Y_c]$ and $t'[Y_b]$, *i.e.,* we cannot match entire $t[Y_c]$ and $t'[Y_b]$ by just comparing their email or phone attributes. Nevertheless, putting them together with the matching key given above, we can infer three new matching keys:

1. LN, FN and phone, via $=, \approx_d, =$ operators, respectively,
2. address and email, to be compared via =, and
3. phone and email, to be compared via =.

These deduced keys have added value. While we cannot match t_1 and t_4–t_6 by using the key given earlier, we can match these tuples based on the deduced keys. Indeed, using key (3), we can now match t_1 and t_6 in Fig. 6.1: they have the same phone and email, and can thus be identified, although their name, gender and address attributes are *radically different*. That is, although there are errors in those attributes, we are still able to match the records by inspecting their email and phone attributes. Similarly we can match t_1 and t_4, and t_1 and t_5 using keys (1) and (2), respectively. □ □

The example highlights the need for effective techniques to specify and reason about the semantics of data in unreliable relations for record matching. One can draw an analogy of this to our familiar notion of functional dependencies (FDs). Indeed, to identify a tuple in a relation we use candidate keys. To find the keys we first specify a set of FDs, and then infer keys by the *implication analysis* of the FDs. For all the reasons that we need FDs and their reasoning techniques for identifying tuples in a clean relation, it is also important to develop (a) dependencies to specify the semantics of data in relations that may contain errors, and (b) effective techniques to reason about these dependencies.

One might be tempted to use FDs in record matching. Unfortunately, FDs and other traditional dependencies are defined on clean (error-free) data, mostly for schema design (see, *e.g.,* [AHV95]). In contrast, for record matching we have to accommodate errors and different representations in different data sources. As will be seen shortly, in this context we need a form of dependencies quite *different* from their traditional counterparts, and a reasoning mechanism more *intriguing* than the standard notion of implication analysis.

The need for dependencies in record matching has long been recognized (*e.g.,* [HS95, ARS09, WNJ+08, CSGK07, SLD05]). It is known that matching keys typically assure *high match accuracy* [EIV07]. However, no previous work has studied how to specify and reason about dependencies for matching records across unreliable data sources.

Contributions. This chapter proposes a class of dependencies for record matching, and provides techniques for reasoning about such dependencies.

(1) Our first contribution is a class of *matching dependencies* (MDs) of the form: if some attributes match then *identify* other attributes. For instance, all the semantic relations we have seen in Example 6.1.1 can be expressed as MDs. In contrast to traditional dependencies, matching dependencies have a *dynamic (update) semantics* to accommodate errors in unreliable data sources. They are defined in terms of *similarity*

operators and across possibly *different relations*.

(2) Our second contribution is a formalization of matching keys, referred to as *relative candidate keys* (RCKs). RCKs are a special class of MDs that match tuples by comparing a *minimum number* of attributes. For instance, the matching keys (1-3) given in Example 6.1.1 are RCKs relative to (Y_c, Y_b). The notion of RCKs substantially differs from traditional candidate keys for relations: they aim to identify tuples across possibly different, unreliable data sources.

(3) Our third contribution is a generic reasoning mechanism for deducing MDs from a set of given MDs. For instance, keys (1-3) of Example 6.1.1 can be deduced from the MDs given there. In light of the dynamic semantics of MDs, the reasoning is a departure from our familiar terrain of traditional dependency implication.

(4) Our fourth contribution is a sound and complete inference system for deducing MDs from a set of given MDs, along the same lines as the Armstrong's Axioms for the implication analysis of FDs (see, *e.g.,* [AHV95]). The inference of MDs is, however, more involved than its FDs counterpart: it consists of nine rules, instead of three axioms.

(5) Our fifth contribution is an algorithm for determining whether an MD can be deduced from a set of MDs. Despite the dynamic semantics of MDs and the use of similarity operators, the deduction algorithm is in $O(n^2)$ time, where n is the size of MDs. This is comparable to the traditional implication analysis of FDs.

(6) Our sixth contribution is an algorithm for deducing a set of RCKs from MDs. Recall that it takes exponential time to enumerate all candidate keys from a set of FDs [LO78]. For the same reason it is unrealistic to compute all RCKs from MDs. To cope with this we introduce a quality model such that for any given number k, the algorithm returns k quality RCKs *w.r.t.* the model, in $O(kn^3)$ time, where n is as above.

We remark that the reasoning is efficient: it is done at the schema level and at compile time, and n is the size of MDs (analogous to the size of FDs), which is typically much smaller than that of data on which matching is conducted.

(7) Our final contribution is an experimental study. We first evaluate the scalability of our reasoning algorithms, and find them quite efficient. For instance, it takes less than 100 seconds to deduce 50 quality RCKs from a set of 2000 MDs. Moreover, we evaluate the impacts of RCKs on the quality and performance of two record matching methods: statistical and rule-based. Using real-life data scraped from the Web, we find that RCKs improve match quality by up to 20%, in terms of *precision* (the ratio of *true* matches correctly found to all matches returned, true or false) and *recall* (the ratio of *true* matches correctly found to all matches in the data, correctly found or

incorrectly missed). In many cases RCKs improve the efficiency as well. In addition, RCKs are also useful in blocking and windowing, two of the widely used optimization techniques for matching records in large relations (see below). We find that blocking and windowing based on (part of) RCKs consistently lead to better match quality, with 10% improvement.

Applications. This work does not aim to introduce another record matching algorithm. It is to *complement* existing methods and to improve their match quality and efficiency when dealing with large, unreliable data sources. In particular, it provides effective techniques to find keys for matching, blocking and windowing.

Matching. Naturally RCKs provide matching keys: they tell us what attributes to compare and how to compare them. As observed in [Jar89], to match tuples of arity n, there are 2^n possible comparison configurations. Thus it is unrealistic to enumerate all matching keys exhaustively and then manually select "the best keys" among possibly exponentially many candidates. In contrast, RCKs are automatically deduced from MDs *at the schema level and at compile time*. In addition, RCKs reduce the cost of inspecting a single pair of tuples by minimizing the number of attributes to compare.

Better still, RCKs improve match quality. Indeed, deduced RCKs *add value*: as we have seen in Example 6.1.1, while tuples t_4–t_6 and t_1 cannot be matched by the given key, they are identified by the deduced RCKs. The added value of deduced rules has long been recognized in census data cleaning: deriving implicit rules from explicit ones is a routine practice of US Census Bureau [FH76, Win04].

Blocking. To handle large relations it is common to partition the relations into blocks based on blocking keys (discriminating attributes), such that only tuples in the same block are compared (see, *e.g.,* [EIV07]). This process is often repeated multiple times to improve match quality, each using a different blocking key. The match quality is highly dependent on *the choice of blocking keys*. As shown by our experimental results, blocking can be effectively done by grouping similar tuples by (*part of*) RCKs.

Windowing. An alternative way to cope with large relations is by first sorting tuples using a key, and then comparing the tuples using a sliding window of a fixed size, such that only tuples within the same window are compared [HS95]. As verified by our experimental study, (*part of*) RCKs suffice to serve as quality sorting keys.

We contend that the MD-based techniques can be readily incorporated into matching tools to improve their quality and efficiency. Provided a small initial set of MDs that are either designed based on domain knowledge or discovered from sample data

(via, *e.g.,* expectation maximization (EM) algorithm [Jar89, Win02]), matching tools can employ the reasoning techniques to automatically derive quality RCKs, and use them as keys for matching, blocking and windowing.

Organization. The remainder of the chapter is organized as follows. Section 6.2 defines MDs and RCKs. Section 6.3 introduces reasoning mechanism and an inference system for MDs. Algorithms for deducing MDs and RCKs are provided in Sections 6.4 and 6.5, respectively. The experimental study is presented in Section 6.6, followed by related work in Section 6.7.

6.2 Matching Dependencies and Relative Candidate Keys

In this section we first define matching dependencies, and then present the notion of relative candidate keys.

6.2.1 Matching Dependencies

Let R_1 and R_2 be two relation schemas, and Y_1 and Y_2 lists of attributes in R_1 and R_2, respectively. The record matching problem is stated as follows.

Given an instance (I_1, I_2) of (R_1, R_2), *the record matching problem* is to identify all tuples $t_1 \in I_1$ and $t_2 \in I_2$ such that $t_1[Y_1]$ and $t_2[Y_2]$ refer to the same real-world entity.

Observe the following. (a) Even when $t_1[Y_1]$ and $t_2[Y_2]$ refer to the same entity, one may still find that $t_1[Y_1] \neq t_2[Y_2]$ due to errors or different representations in the data. (b) The problem aims to match $t_1[Y_1]$ and $t_2[Y_2]$, *i.e.,* parts of t_1 and t_2 specified by lists of attributes, not necessarily the entire tuples t_1 and t_2. (c) It is to find matches across relations of possibly different schemas.

To accommodate these in record matching we define MDs in terms of similarity operators and a notion of comparable lists, a departure from our familiar FDs. Before we define MDs, we first present these notions. To simplify the discussion, we assume that R_1 and R_2 specify distinct data sources. Nevertheless, all the results in this work remain intact the context where R_1 and R_2 denote the same relation.

Similarity operators. Assume a fixed set Θ of domain-specific similarity relations. For each $\approx$ in Θ, and values x, y in the specific domains in which $\approx$ is defined, we write $x \approx y$ if (x, y) is in $\approx$, and refer to $\approx$ as a *similarity operator*. The operator can be any

similarity metric used in record matching, *e.g.,* q-grams, Jaro distance or edit distance (see [EIV07] for a survey), such that $x \approx y$ is true if x and y are "close" enough *w.r.t.* a predefined threshold.

In particular, the equality relation $=$ is in Θ.

We also use a *matching operator* $\rightleftharpoons$: for any values x and y, $x \rightleftharpoons y$ indicates that x and y are identified via updates, *i.e.,* we update x and y to make them identical. The semantics of the operator $\rightleftharpoons$ will be elaborated shortly.

Comparable lists. For a list X of attributes in a schema R, we denote the length of X by $|X|$, and the i-th element of X by $X[i]$. We use $A \in R$ (resp. $A \in X$) to denote that A is an attribute in R (resp. X), and use $\mathsf{dom}(A)$ to denote its domain.

A pair of lists (X_1, X_2) are said to be *comparable* over (R_1, R_2) if (a) X_1 and X_2 are of the same length, and (b) their elements are *pairwise comparable*, *i.e.,* for each $j \in [1, |X_1|]$, $X_1[j] \in R_1$, $X_2[j] \in R_2$, and $\mathsf{dom}(X_1[j]) = \mathsf{dom}(X_2[j])$ (to simplify the discussion, we assume *w.l.o.g.* that $X_1[j]$ and $X_2[j]$ have the same domain, which can be achieved by data standardization; see [EIV07] for details). We write $(X_1[j], X_2[j]) \in (X_1, X_2)$ for $j \in [1, |X_1|]$.

Matching dependencies. A *matching dependency* (MD) φ for (R_1, R_2) is syntactically defined as follows:

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

where (1) (X_1, X_2) (resp. (Z_1, Z_2)) are comparable lists over (R_1, R_2), and (2) for each $j \in [1, k]$, $\approx_j$ is a similarity operator in Θ, and $k = |X_1|$.

We refer to $\bigwedge_{j \in [1,k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]])$ and $(R_1[Z_1], R_2[Z_2])$ as the LHS and RHS of φ, respectively.

Intuitively, φ states that if $R_1[X_1]$ and $R_2[X_2]$ are similar *w.r.t.* some similarity metrics, then $R_1[Z_1]$ and $R_2[Z_2]$ refer to the same object and should be identified (made identical).

Example 6.2.1: The semantic relations given in Examples 6.1.1 can be expressed as MDs, as follows:

$$\begin{aligned}
\varphi_1\colon\ & \mathsf{credit}[\mathrm{LN}] = \mathsf{billing}[\mathrm{LN}] \wedge \mathsf{credit}[\mathsf{addr}] = \mathsf{billing}[\mathsf{post}] \wedge \\
& \quad \mathsf{credit}[\mathrm{FN}] \approx_d \mathsf{billing}[\mathrm{FN}] \ \rightarrow\ \mathsf{credit}[Y_c] \rightleftharpoons \mathsf{billing}[Y_b] \\
\varphi_2\colon\ & \mathsf{credit}[\mathsf{tel}] = \mathsf{billing}[\mathsf{phn}] \rightarrow \mathsf{credit}[\mathsf{addr}] \rightleftharpoons \mathsf{billing}[\mathsf{post}] \\
\varphi_3\colon\ & \mathsf{credit}[\mathsf{email}] = \mathsf{billing}[\mathsf{email}] \rightarrow \mathsf{credit}[\mathrm{FN}, \mathrm{LN}] \rightleftharpoons \mathsf{billing}[\mathrm{FN}, \mathrm{LN}]
\end{aligned}$$

where φ_1 states that for any credit tuple t and billing tuple t', if t and t' have the same last name and address, and if their first names are similar *w.r.t.* $\approx_d$ (but may not necessarily

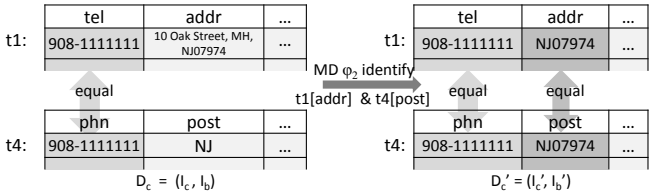


Figure 6.2: MDs expressing matching rules

be identical), then $t[Y_c]$ and $t'[Y_b]$ should be identified. Similarly, if t and t' have the same phone number then we should identify their addresses (φ_2); and if t and t' have the same email then their names should be identified (φ_3). Note that while name, address and phone are part of Y_b and Y_c, email is *not*, *i.e.,* the LHS of an MD is neither necessarily contained in nor disjoint from its RHS. □

Dynamic semantics. Recall that a functional dependency (FD) $X \to Y$ simply assures that for any tuples t_1 and t_2, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. In contrast, to accommodate unreliable data, the semantics of MDs is more involved. To present the semantics we need the following notations.

Extensions. To keep track of tuples during a matching process, we assume a temporary unique tuple id for each tuple. For instances I and I' of the same schema, we write $I \sqsubseteq I'$ if for each tuple t in I there is a tuple t' in I' such that t and t' have the same tuple id. Here t' is an updated version of t, and t' and t may differ in some attribute values. For two instances $D = (I_1, I_2)$ and $D' = (I_1', I_2')$ of (R_1, R_2), we write $D \sqsubseteq D'$ if $I_1 \sqsubseteq I_1'$ and $I_2 \sqsubseteq I_2'$.

For tuples $t_1 \in I_1$ and $t_2 \in I_2$, we write $(t_1, t_2) \in D$.

LHS matching. We say that $(t_1, t_2) \in D$ *match* the LHS of MD φ if for each $j \in [1, k]$, $t_1[X_1[j]] \approx_j t_2[X_2[j]]$, *i.e.,* $t_1[X_1[j]]$ and $t_2[X_2[j]]$ are similar *w.r.t.* the metric $\approx_j$.

For example, t_1 and t_3 of Fig. 6.1 match the LHS of φ_1 of Example 6.2.1: t_1 and t_3 have identical LN and address, and "Mark" $\approx_d$ "Marx" when $\approx_d$ is an edit distance metric.

Semantics. We are now ready to give the semantics. Consider a pair (D, D') of instances of (R_1, R_2), where $D \sqsubseteq D'$.

The pair (D, D') of instances *satisfy* MD φ, denoted by $(D, D') \models \varphi$, if for any tuples $(t_1, t_2) \in D$, if (t_1, t_2) match the LHS of φ in the instance D, then *in the other instance* D', (a) $t_1[Z_1] = t_2[Z_2]$, *i.e.,* the RHS attributes of φ in t_1 and t_2 are identified; and (b) (t_1, t_2) also match the LHS of φ.

Intuitively, the semantics states how φ is *enforced* as a *matching rule*: whenever (t_1, t_2) in an instance D match the LHS of φ, $t_1[Z_1]$ and $t_2[Z_2]$ ought to be made equal.

The outcome of the enforcement is reflected in the other instance D'. That is, some value V is to be found such that $t_1[Z_1] = V$ and $t_1[Z_2] = V$ in D', although V is not explicitly specified.

Example 6.2.2: Consider the MD φ_2 of Example 6.2.1 and the instance $D_c = (I_c, I_b)$ of Fig. 6.1, in which (t_1, t_4) match the LHS of φ_2. As depicted in Fig. 6.2, the enforcement of φ_2 yields another instance $D'_c = (I'_c, I'_b)$ in which $t_1[\mathsf{addr}] = t_4[\mathsf{post}]$, while $t_1[\mathsf{addr}]$ and $t_4[\mathsf{post}]$ are *different* in D_c.

The $\rightleftharpoons$ operator only requires that $t_1[\mathsf{addr}]$ and $t_4[\mathsf{post}]$ are identified, but does not specify how they are updated. That is, in any D'_c that extends D_c, if (a) $t_1[\mathsf{addr}] = t_4[\mathsf{post}]$ and (t_1, t_4) match the LHS of φ_2 in D'_c, and (b) similarly for $t_1[\mathsf{addr}]$ and $t_6[\mathsf{post}]$ are identified in D'_c, then φ_2 is considered enforced on D'_c, *i.e.,* $(D_c, D'_c) \models \varphi_2$. □

It should be clarified that we use updates just to give the semantics of MDs. In the matching process instance D may *not* be updated, *i.e.,* there is *no* destructive impact on D.

Matching dependencies (MDs) are quite *different* from traditional dependencies such as FDs.

- MDs have "dynamic" semantics to accommodate errors and different representations in the data: if attributes $t_1[X_1]$ and $t_2[X_2]$ match in instance D, then $t_1[Z_1]$ and $t_2[Z_2]$ are updated and identified. Here $t_1[Z_1]$ and $t_2[Z_2]$ are equal in *another instance* D' that results from the updates to D, although they may be *radically different* in the original instance D. In contrast, FDs have a "static" semantics: if certain attributes are equal in D, then some other attributes must be equal in *the same* instance D.
- MDs are defined with *similarity metrics* and the matching operator $\rightleftharpoons$, whereas FDs are defined with equality only.
- MDs are defined across possibly different relations, while FDs are defined on a single relation.

Example 6.2.3: Consider two FDs defined on schema $R(A, B, E)$:

$$f_1\colon A \to B, \qquad f_2\colon B \to E.$$

Consider instances I_0 and I_1 of R shown in Fig. 6.3. Then s_1 and s_2 in I_0 violate f_1: $s_1[A] = s_2[A]$ but $s_1[B] \neq s_2[B]$; similarly, s_1 and s_2 in I_1 violate f_2.

In contrast, consider two MDs defined on R:

$$\psi_1\colon R[A] = R[A] \to R[B] \rightleftharpoons R[B],$$
$$\psi_2\colon R[B] = R[B] \to R[E] \rightleftharpoons R[E],$$

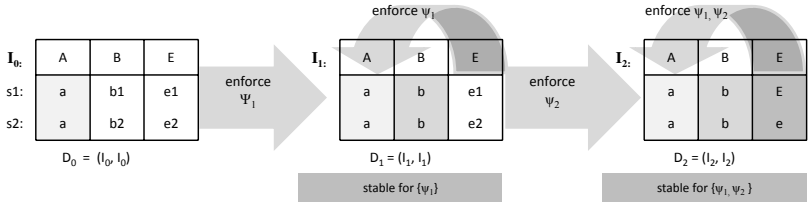


Figure 6.3: The dynamic semantics of MDs

where ψ_1 states that for any (s_1, s_2), if $s_1[A] = s_2[A]$, then $s_1[B]$ and $s_2[B]$ should be identified; similarly for ψ_2.

Let $D_0 = (I_0, I_0)$ and $D_1 = (I_1, I_1)$. Then $(D_0, D_1) \models \psi_1$. While $s_1[A] = s_2[A]$ but $s_1[B] \neq s_2[B]$ in I_0, s_1 and s_2 are *not* treated a violation of ψ_1. Instead, a value b is found such that $s_1[B]$ and $s_2[B]$ are changed to b, which results in instance I_1. This is how MDs accommodate errors in unreliable data sources. Note that $(D_0, D_1) \models \psi_2$ since $s_1[B] \neq s_2[B]$ in I_0, *i.e.,* (s_1, s_2) does not match the LHS of ψ_2 in I_0. □

A pair (D, D') of instances *satisfy* a set Σ of MDs, denoted by $(D, D') \models \Sigma$, if $(D, D') \models \varphi$ for all $\varphi \in \Sigma$.

6.2.2 Relative Candidate Keys

To decide whether $t_1[Y_1]$ and $t_2[Y_2]$ refer to the same entity, it is natural to consider a minimal number of attributes to compare. In light of this, we identify a special case of MDs.

A *key* ψ *relative to* attribute lists (Y_1, Y_2) of (R_1, R_2) is an MD in which the RHS is fixed to be (Y_1, Y_2), *i.e.,* an MD of the form $\bigwedge_{j \in [1,k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrow R_1[Y_1] \rightleftharpoons R_2[Y_2]$, where $k = |X_1| = |X_2|$. We simply write ψ as

$$(X_1, X_2, C), \quad \text{where } C = [\approx_1, \ldots, \approx_k],$$

when (Y_1, Y_2) is clear from the context. We refer to k as the *length* of ψ, and C as its *comparison vector*.

The key ψ assures that for any tuples (t_1, t_2) of (R_1, R_2), to identify $t_1[Y_1]$ and $t_2[Y_2]$ it suffices to inspect whether the attributes of $t_1[X_1]$ and $t_2[X_2]$ pairwise match *w.r.t.* C.

The key ψ is a relative *candidate* key (RCK) if there is no other key $\psi' = (X_1', X_2', C')$ relative to (Y_1, Y_2) such that (1) the length l of ψ' is less than the length k of key ψ, and (2) for each $i \in [1, l]$, $X_1'[i], X_2'[i]$ and $C'[i]$ are the j-th element of the lists X_1, X_2 and C, respectively, for some $j \in [1, k]$.

We write $\psi' \preceq \psi$ if conditions (1) and (2) are satisfied.

Intuitively, ψ is a RCK if no other key ψ' relative to (Y_1, Y_2) requires less attributes to inspect. To identify $t_1[Y_1]$ and $t_2[Y_2]$, an RCK specifies a *minimum* list of attributes to

inspect and tells us *how* to compare these attributes.

Example 6.2.4: Candidate keys relative to (Y_c, Y_b) include:

rck_1: ([LN, addr, FN], [LN, post, FN], $[=, =, \approx_d]$)

rck_2: ([LN, tel, FN], [LN, phn, FN], $[=, =, \approx_d]$)

rck_3: ([email, addr], [email, post], $[=, =]$)

rck_4: ([email, tel], [email, phn], $[=, =]$)

Here the key rck_4 states that for any credit tuple t and any billing tuple t', if t[email, tel] = t'[email, phn], then $t[Y_c]$ and $t'[Y_b]$ match; similarly for rck_1, rck_2 and rck_3. We also remark that email is not part of Y_b or Y_c. □

One can draw an analogy of RCKs to the familiar notion of keys for relations: both notions attempt to provide an invariant connection between tuples and the real-world entities they represent. However, there are *sharp differences* between the two notions. First, RCKs bring domain-specific *similarity* operators into the play, carrying a *comparison vector*. Second, RCKs are defined *across different relations*; in contrast, keys are defined on a single relation. Third, RCKs have a *dynamic semantics* and aim to identify *unreliable* data, a departure from the classical dependency theory.

6.3 Reasoning about Matching Dependencies

Implication analysis of FDs can be found in almost every database textbook. Along the same lines we naturally want to deduce MDs from a set of given MDs. However, as opposed to traditional dependencies, MDs are defined in terms of domain-specific similarity and matching operators, and they have dynamic semantics. As a result, traditional implication analysis no longer works for MDs.

Below we first propose a generic mechanism to deduce MDs, independent of any particular similarity operators. We then present a sound and complete inference system for MDs, which provide algorithmic insight into deducing MDs.

6.3.1 A Generic Reasoning Mechanism

A new challenge encountered when reasoning about MDs involves similarity operators in MDs, which may not be themselves expressible in any logic formalism. In light of these, our reasoning mechanism is necessarily generic.

Generic axioms. We assume only *generic axioms* for each similarity operator $\approx$ in Θ as follows.

- It is reflexive, *i.e.,* $x \approx x$.
- It is symmetric, *i.e.,* if $x \approx y$ then $y \approx x$.
- It subsumes equality, *i.e.,* if $x = y$ then $x \approx y$.

Nevertheless, except equality =, $\approx$ is *not* assumed transitive in general, *i.e.,* from $x \approx y$ and $y \approx z$ it does *not* necessarily follow that $x \approx z$.

The equality relation = is reflexive, symmetric and transitive, as usual. In addition, for any similarity operator $\approx$ and values x and y, if $x \approx y$ and $y = z$, then $x \approx z$.

To simplify the discussion we also assume the following. (1) There is a unique similarity operator $\approx_A$ defined on each distinct (infinite) domain $\mathsf{dom}(A)$. (2) The similarity operator $\approx_A$ is *dense*: for any number k, there exist values $v, v_1, \ldots, v_k \in \mathsf{dom}(A)$ such that $v \approx_A v_i$ for $i \in [1,k]$, and $v_i \not\approx_A v_j$ for all $i, j \in [1,k]$ and $i \neq j$. That is, there are unboundedly many distinct values that are within a certain distance *w.r.t.* $\approx_A$, but are not similar to each other.

Many similarity operators commonly found in practice are dense, *e.g.,* edit distance. However, the linear ordering in a numeric domain may not be dense. As will be seen shortly, the proofs of Section 6.3.2 leverage the density to avoid discussions on subtle issues raised by similarity operators.

The limitations of implication analysis. Another challenge is posed by the dynamic semantics of MDs. Recall the notion of implication (see, *e.g.,* [AHV95]): given a set Γ of traditional dependencies and another dependency ϕ, Γ *implies* ϕ if for any database D that satisfies Γ, D also satisfies ϕ. For an example of our familiar FDs, if Γ consists of $X \to Y$ and $Y \to Z$, then it implies $X \to Z$. However, this notion of implication is no longer applicable to MDs on unreliable data, as illustrated below.

Example 6.3.1: Let Σ_0 be the set $\{\psi_1, \psi_2\}$ of MDs and Γ_0 the set $\{f_1, f_2\}$ of FDs given in Example 6.2.3. Consider additional MD and FD given below:

MD ψ_3: $R[A] = R[A] \to R[E] \rightleftharpoons R[E]$,
FD f_3: $A \to E$.

Then Γ_0 implies f_3, but Σ_0 *does not* imply ψ_3. To see this, consider I_0 (D_0) and I_1 (D_1) in Fig. 6.3. Observe the following.
(1) $(D_0, D_1) \models \Sigma_0$ but $(D_0, D_1) \not\models \psi_3$. Indeed, $(D_0, D_1) \models \psi_1$ and $(D_0, D_1) \models \psi_2$. However, $(D_0, D_1) \not\models \psi_3$: while $s_1[A] = s_2[A]$ in D_0, $s_1[E] \neq s_2[E]$ in D_1. This tells us that Σ_0 does *not* imply ψ_3 if the notion of implication is used for MDs.
(2) In contrast, neither I_0 nor I_1 contradicts to the implication of f_3 from Γ_0. Note that $I_0 \not\models f_3$: $s_1[A] = s_2[A]$ but $s_1[E] \neq s_2[E]$. That is, s_1 and s_2 violate f_3. However, I_0

does not satisfy Γ_0 either. Indeed, $I_0 \not\models f_1$: $s_1[A] = s_2[A]$ but $s_1[B] \neq s_2[B]$. Thus the conventional implication of FDs remains valid on I_0; similarly for I_1. □

Deduction. To capture the dynamic semantics of MDs in the deduction analysis, we need the following notion.

An instance D of (R_1, R_2) is said to be *stable* for a set Σ of MDs if $(D, D) \models \Sigma$. Intuitively, a stable instance D is an ultimate outcome of enforcing Σ: each and every rule in Σ is enforced until no more updates have to be conducted.

Example 6.3.2: As illustrated in Fig. 6.3, D_2 is a stable instance for Σ_0 of Example 6.3.1. It is an outcome of enforcing MDs in Σ_0 as matching rules: when ψ_1 is enforced on D_0, it yields another instance in which $s_1[B] = s_2[B]$, *e.g.,* D_1. When ψ_2 is further enforced on D_1, $s_1[E]$ and $s_2[E]$ are identified, yielding D_2. Now $(D_2, D_2) \models \Sigma_0$, *i.e.,* no further changes are necessary for enforcing the MDs in Σ_0. □

We are now ready to formalize the notion of deductions.

For a set Σ of MDs and another MD φ on (R_1, R_2), φ is said to be *deduced* from Σ, denoted by $\Sigma \models_m \varphi$, if for any instance D of (R_1, R_2), and *for each* stable instance D' for Σ, if $(D, D') \models \Sigma$ then $(D, D') \models \varphi$.

Intuitively, stable instance D' is a "fixpoint" reached by enforcing Σ on D. There are possibly many such stable instances, depending on how D is updated. The deduction analysis inspects *all* of the stable instances for Σ.

The notion of deductions is generic: no matter how MDs are interpreted, if Σ is enforced, then so must be φ. In other words, φ is a *logical consequence* of the given MDs in Σ.

Example 6.3.3: As will be seen in Section 6.3.2, for Σ_0 and ψ_3 given in Example 6.3.1, $\Sigma_0 \models_m \psi_3$. In particular, for the instance D_0 and the stable instance D_2 of Example 6.3.2, one can see that $(D_0, D_2) \models \Sigma_0$ and $(D_0, D_2) \models \psi_3$. □

The *deduction problem* for MDs is to determine, given any set Σ of MDs defined on (R_1, R_2) and another MD φ defined on (R_1, R_2), whether $\Sigma \models_m \varphi$.

Added value of deduced MDs. While the dynamic semantics of MDs makes it difficult to reason about MDs, it yields *added value* of deduced MDs. Indeed, while tuples in unreliable relations may not be matched by a given set Σ of MDs, they may be identified by an MD φ deduced from Σ. In contrast, when a traditional dependency ϕ is *implied* by a set of dependencies, any database that violates ϕ cannot possibly satisfy all the given dependencies.

Example 6.3.4: Let D_c be the instance of Fig. 6.1, and Σ_1 consist of $\varphi_1, \varphi_2, \varphi_3$ of

Example 6.2.1. As shown in Example 6.1.1, (t_1, t_6) in D_c can be matched by rck_4 of Example 6.2.4, but cannot be directly identified by Σ_1. Indeed, one can easily find an instance D' such that $(D_c, D') \models \Sigma_1$ but $t_1[Y_c] \neq t_6[Y_b]$ in D'. In contrast, there is no D' such that $(D_c, D') \models \mathsf{rck}_4$ but $t_1[Y_c] \neq t_6[Y_b]$ in D'. As will be seen in Example 6.3.5, it is from Σ_1 that rck_4 is deduced. This shows that while tuples may not be matched by a set Σ of given MDs, they can be identified by MDs deduced from Σ.

The deduced rck_4 would not have had added value if the MDs were interpreted with a static semantics like FDs. Indeed, the tuples t_1 and t_6 have *radically different* names and addresses, and would be considered as a violation of rck_4 if rck_4 were treated as an "FD". At the same time they would violate the φ_1 in Σ_1. This tells us that with the conventional implication analysis, rck_4 would not be able to identify tuples that Σ_1 fails to match. □

6.3.2 A Sound and Complete Inference System for MDs

Armstrong's Axioms have proved extremely useful in the implication analysis of FDs (see, *e.g.,* [AHV95]). Along the same lines one naturally wants a finite inference system that is *sound and complete* for the deduction analysis of MDs.

The inference of MDs is, however, more involved than its FDs counterpart. (1) The matching operator $\rightleftharpoons$ updates data to identify data elements. It interacts with equality $=$: $u \rightleftharpoons v$ entails that $u = v$ in the updated data. (2) Similarity metrics also interact with equality $=$, *e.g.,* if $u \approx v$ then $u = v$, and if $u = v$ and $v \approx w$ then $u \approx w$. (3) MDs are defined on lists of attributes, whereas FDs are defined on sets of attributes.

Weak MDs. To capture the interactions in the deduction analysis, we introduce a weak form of MDs to express intermediate results encountered in the inference. A weak MD allows similarity metrics to appear in the RHS, in contrast to the matching operator $\rightleftharpoons$ as found in MDs. More specifically, A *weak* MD over (R_1, R_2) is of the form:

$$\varphi_1 = \bigwedge_{j\in[1,k]}(R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrowtail R_1[A] \approx R_2[B],$$
$$\varphi_2 = \bigwedge_{j\in[1,k]}(R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrowtail R_i[A] \approx' R_i[B],$$

where $i \in [1,2]$ (*i.e.,* in φ_2, the RHS may refer to the same relation R_1 or R_2), and $\approx$ and $\approx'$ are similarity operators in Θ. we use $\rightarrowtail$ instead of $\rightarrow$ to explicitly distinguish weak MDs from MDs.

The semantics of weak MDs is a variation of its MD counterpart. Let (D, D') be a pair of instances of (R_1, R_2), where $D \sqsubseteq D'$. The pair (D, D') of instances *satisfy weak*

MD$_1$: If $\bigwedge_{i\in[1,\,k]}(R_2[X_2[i]] \approx_i R_1[X_1[i]]) \rightarrow R_2[Z_2] \rightleftharpoons R_1[Z_1]$, then $\bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$.

MD$_2$: Let $L = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]])$.

For each $i \in [1,k]$, (a) if $\approx_i$ is $=$, then $L \rightarrow R_1[X_1[i]] \rightleftharpoons R_2[X_2[i]]$, and (b) $L \rightarrowtail R_1[X_1[i]] \approx_i R_2[X_2[i]]$ otherwise.

MD$_3$: If $\phi = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$, then for any comparable attributes (A,B) over (R_1,R_2) and similarity operator $\approx$ in Θ, (a) $\mathsf{LHS}(\phi)\bigwedge(R_1[A] \approx R_2[B]) \rightarrow \mathsf{RHS}(\phi)$, and (b) $\mathsf{LHS}(\phi)\bigwedge(R_1[A] = R_2[B]) \rightarrow \mathsf{RHS}(\phi)\bigwedge(R_1[A] \rightleftharpoons R_2[B])$.

MD$_4$: Let $L = \bigwedge_{j\in[1,\,g]}(R_1[Y_1[j]] \approx_j R_2[Y_2[j]])$ such that for each $j \in [1,\,g]$, $\approx_j$ is not $=$.

If (a) $\phi_1 = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[W_1] \rightleftharpoons R_2[W_2]$, (b) $\phi_2 = L\bigwedge(R_1[W_1] = R_2[W_2]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$, and (c) for each $j \in [1,g]$, $\mathsf{LHS}(\phi_1) \rightarrowtail R_1[Y_1[j]] \approx_j R_2[Y_2[j]]$, then $\mathsf{LHS}(\phi_1)\bigwedge L \rightarrow \mathsf{RHS}(\phi_2)$.

MD$_5$: Let $L = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]])$.

If $\phi = L\bigwedge(R_1[A] \approx R_2[B]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$, then $L\bigwedge(R_1[A] = R_2[B]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$.

MD$_6$: If $\phi_1 = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[E_1E_2] \rightleftharpoons R_2[FF]$, then $\mathsf{LHS}(\phi_1) \rightarrowtail R_1[E_1] = R_1[E_2]$;

If $\phi_2 = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[EE] \rightleftharpoons R_2[F_1F_2]$, then $\mathsf{LHS}(\phi_2) \rightarrowtail R_2[F_1] = R_2[F_2]$.

MD$_7$: Let $L = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]])$.

If $L \rightarrowtail R_1[A_1] \approx R_2[B_1]$ and $L \rightarrow R_1[A_2] \rightleftharpoons R_2[B_1]$, then $L \rightarrowtail R_1[A_1] \approx R_1[A_2]$;

If $L \rightarrowtail R_1[A_1] \approx R_2[B_1]$ and $L \rightarrow R_1[A_1] \rightleftharpoons R_2[B_2]$, then $L \rightarrowtail R_2[B_1] \approx R_2[B_2]$.

MD$_8$: Let $\phi = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[E_1] \rightleftharpoons R_2[F_1]$.

If ϕ and $\mathsf{LHS}(\phi) \rightarrowtail R_1[E_1] \approx R_1[E_2]$, then (a) $\mathsf{LHS}(\phi) \rightarrow R_1[E_2] \rightleftharpoons R_2[F_1]$ if $\approx$ is $=$, and (b) $\mathsf{LHS}(\phi) \rightarrowtail R_1[E_2] \approx R_2[F_1]$ otherwise;

If ϕ and $\mathsf{LHS}(\phi) \rightarrowtail R_2[F_1] \approx R_2[F_2]$, then (a) $\mathsf{LHS}(\phi) \rightarrow R_1[E_1] \rightleftharpoons R_2[F_2]$ if $\approx$ is $=$, and (b) $\mathsf{LHS}(\phi) \rightarrowtail R_1[E_1] \approx R_2[F_2]$ otherwise.

MD$_9$: Let $L = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]])$.

If $L \rightarrowtail R_1[E_1] = R_1[E_2]$ and $L \rightarrowtail R_1[E_1] \approx R_2[F_1]$, then $L \rightarrowtail R_1[E_2] \approx R_2[F_1]$;

If $L \rightarrowtail R_2[F_1] = R_2[F_2]$ and $L \rightarrowtail R_1[E_1] \approx R_2[F_1]$, then $L \rightarrowtail R_1[E_1] \approx R_2[F_2]$.

Figure 6.4: Inference System $\mathcal{I}$ for MDs

MD φ_1, denoted by $(D,D') \models \varphi_1$, if for any tuples $(t_1,t_2) \in D$, if (t_1,t_2) match the LHS of φ_1 in D, then in D', $t_1[A] \approx t_2[B]$ and moreover, (t_1,t_2) also match the LHS of φ_1.

Similarly, $(D,D') \models \varphi_2$ if for any tuples $(t_1,t_2) \in D$, if (t_1,t_2) match the $\mathsf{LHS}(\varphi_2)$

in D, then in D', $t_i[A] \approx t_i[B]$, where t_i is either t_1 or t_2, and (t_1, t_2) also match $\mathsf{LHS}(\varphi_2)$.

An inference system. Using MDs and weak MDs, we propose the inference system $\mathcal{I}$ in Fig. 6.4. It consists of nine axioms MD_1–MD_9.

- MD_1 reveals the "symmetricity" of MDs: the order of relations R_1 and R_2 in an MD can be swapped.
- MD_2, MD_3 and MD_4 extend the reflexivity, augmentation and transitivity rules of the Armstrong's axioms for FDs, respectively
- MD_5 states that a similarity operator $\approx$ in the LHS of an MD can be *upgraded* to equality $=$ since $\approx$ subsumes $=$.
- MD_6, MD_7 and MD_8 characterize the interactions between the matching operator and similarity metrics, in particular equality; note that MD_6 and MD_7 derive weak MDs, while MD_8 deduces standard MDs.
- MD_9 reveals the interaction between similarity metrics and equality. It derives weak MDs from weak MDs.

Given a set Σ of MDs and another MD φ, we use $\Sigma \vdash_{\mathcal{I}} \varphi$ to denote that φ is provable from Σ using rules in $\mathcal{I}$.

Example 6.3.5: Consider Σ_c consisting of $\varphi_1, \varphi_2, \varphi_3$ of Example 6.2.1, and rck_4 of Example 6.2.4. Then $\Sigma_c \vdash_{\mathcal{I}} \mathsf{rck}_4$ as follows.

(a) $\mathsf{credit}[\mathsf{tel}] = \mathsf{billing}[\mathsf{phn}] \wedge \mathsf{credit}[\mathsf{email}] = \mathsf{billing}[\mathsf{email}]$
$\rightarrow \mathsf{credit}[\mathsf{addr}, \mathsf{email}] \rightleftharpoons \mathsf{billing}[\mathsf{post}, \mathsf{email}]$, ($\psi_1$, by applying MD_3 to φ_2)

(b) $\mathsf{credit}[\mathsf{addr}] = \mathsf{billing}[\mathsf{post}] \wedge \mathsf{credit}[\mathsf{email}] = \mathsf{billing}[\mathsf{email}]$
$\rightarrow \mathsf{credit}[\mathsf{addr}, \mathrm{FN}, \mathrm{LN}] \rightleftharpoons \mathsf{billing}[\mathsf{post}, \mathrm{FN}, \mathrm{LN}]$, ($\psi_2$, by applying MD_3 to φ_3)

(c) $\mathsf{credit}[\mathsf{tel}] = \mathsf{billing}[\mathsf{phn}] \wedge \mathsf{credit}[\mathsf{email}] = \mathsf{billing}[\mathsf{email}]$
$\rightarrow \mathsf{credit}[\mathsf{addr}, \mathrm{FN}, \mathrm{LN}] \rightleftharpoons \mathsf{billing}[\mathsf{post}, \mathrm{FN}, \mathrm{LN}]$, ($\psi_3$, by applying MD_4 to ψ_1 and ψ_2)

(d) $\mathsf{credit}[\mathrm{LN}] = \mathsf{billing}[\mathrm{LN}] \wedge \mathsf{credit}[\mathsf{addr}] = \mathsf{billing}[\mathsf{post}] \wedge \mathsf{credit}[\mathrm{FN}] = \mathsf{billing}[\mathrm{FN}]$
$\rightarrow \mathsf{credit}[Y_c] \rightleftharpoons \mathsf{billing}[Y_b]$, ($\psi_4$, by applying MD_5 to φ_1)

(e) $\mathsf{credit}[\mathsf{tel}] = \mathsf{billing}[\mathsf{phn}] \wedge \mathsf{credit}[\mathsf{email}] = \mathsf{billing}[\mathsf{email}]$
$\rightarrow \mathsf{credit}[Y_c] \rightleftharpoons \mathsf{billing}[Y_b]$, ($\mathsf{rck}_4$, by applying MD_4 to ψ_3 and ψ_4)

Similarly, rck_1, rck_2 and rck_3 can be deduced from Σ_c. □

The inference system $\mathcal{I}$ is sound and complete for the deduction analysis of MDs. That is, for any set Σ of MDs and another MD φ, $\Sigma \models_m \varphi$ iff $\Sigma \vdash_{\mathcal{I}} \varphi$, when the generic reasoning mechanism defined in Section 6.3.1 is concerned. That is, it only assumes *the generic axioms* given there for similarity operators and for equality, regardless of other properties of various domain-specific similarity metrics.

Theorem 6.3.1: *The inference system $\mathcal{I}$ is sound and complete for the deduction analysis of* MDs. □

In the rest of the section we prove Theorem 6.3.1. More specifically, we show that $\mathcal{I}$ is (a) sound and (b) complete for MDs in Lemmas 6.3.2 and 6.3.3, respectively.

Lemma 6.3.2: *Rules* $\mathsf{MD}_1 - \mathsf{MD}_9$ *in the inference system $\mathcal{I}$ are sound for the deduction analysis of* MDs. □

Proof: We show that for any set Σ of MDs and another MD φ over R_1 and R_2, if $\Sigma \vdash_{\mathcal{I}} \varphi$, then $\Sigma \models_m \varphi$. That is, for any instance $D = (I_1, I_2)$ of (R_1, R_2) and for each *stable* instance $D' = (I_1', I_2')$ of D for Σ, if $(D, D') \models \Sigma$ then $(D, D') \models \varphi$.

It suffices to show that each rule in $\mathcal{I}$ is correct, which corresponds to a single step in an inference process. For if it holds, then an induction on the length of proofs using $\mathcal{I}$ can readily verify that $\mathcal{I}$ is sound.

$\underline{\mathsf{MD}_1}$: Let $\phi = \bigwedge_{i\in[1,\,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$ and $\phi^r = \bigwedge_{i\in[1,\,k]}(R_2[X_2[i]] \approx_i R_1[X_1[i]]) \rightarrow R_2[Z_2] \rightleftharpoons R_1[Z_1]$. If $(D, D') \models \phi$, then obviously $(D, D') \models \phi^r$.

$\underline{\mathsf{MD}_2}$*:* This rule extends the reflexivity rule of Armstrong's axioms, by distinguishing two cases: one for equality and the other for non-equality similarity operators. The correctness follows from the definitions of MDs and weak MDs.

$\underline{\mathsf{MD}_3}$*:* This is an extension of the augmentation rule of Armstrong's axioms. That is, one can augment $\mathsf{LHS}(\phi)$ with additional similarity test $R_1[A] \approx R_2[B]$. In particular, if $\approx$ is equality $=$, then $\mathsf{RHS}(\phi)$ can also be expanded accordingly. In contrast to their FD counterpart, the augmentation axioms for MDs have to treat equality and the other similarity operators separately. The correctness of these rules again follows from the definitions of MDs and weak MDs.

$\underline{\mathsf{MD}_4}$: This is the transitivity rule for MDs. To see that it is sound, consider a pair (D, D') of instances such that (a) $(D, D') \models \phi_1$, (b) for each $j \in [1, g]$, $(D, D') \models \mathsf{LHS}(\phi_1) \rightarrowtail R_1[Y_1[j]] \approx_j R_2[Y_2[j]]$, (c) $(D, D') \models \phi_2$, and (d) D' is a stable instance for the given (weak) MDs.

For any two tuples $(t_1, t_2) \in D$, if they match $\mathsf{LHS}(\phi_1)$ and L, then in D', $t_1[W_1] = t_2[W_2]$ by (a), and $t_1[Y_1[j]] \approx_j t_2[Y_2[j]]$ by (b). In addition, (t_1, t_2) match $\mathsf{LHS}(\phi_2)$ in D' by (d). From these it follows that $t_1[Z_1] = t_2[Z_2]$ in D' by (c) and (d), and hence, $\mathsf{LHS}(\phi_1) \wedge L \rightarrow \mathsf{RHS}(\phi_2)$.

$\underline{\mathsf{MD}_5}$*:* Consider instances (D, D') such that $(D, D') \models \phi$. For any tuples $(t_1, t_2) \in D$,

if $t_1[A] = t_2[B]$, then $t_1[A] \approx t_2[B]$ in D. From this it follows that if $(D, D') \models \phi$, then $(D, D') \models L \bigwedge (R_1[A] = R_2[B]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$.

MD_6: Consider instances (D, D') such that $(D, D') \models \phi_1$. For any tuples $(t_1, t_2) \in D$, if they match the LHS of ϕ_1, then $t_1[E_1 E_2] = t_2[FF]$ in D'. Therefore, $t_1[E_1] = t_1[E_2]$. Hence if $(D, D') \models \phi_1$, then $(D, D') \models \mathsf{LHS}(\phi_1) \rightarrowtail R_1[E_1] = R_1[E_2]$.

Similarly, if $(D, D') \models \phi_2$, then $(D, D') \models \mathsf{LHS}(\phi_2) \rightarrowtail R_2[F_1] = R_2[F_2]$.

MD_7, MD_8 *and* MD_9: The soundness of these rules can be verified along the same lines as for MD_6. □

Lemma 6.3.3: *Rules* MD_1–MD_9 *in the inference system* $\mathcal{I}$ *are complete for the deduction analysis of* MDs. □

Proof: We show that for a set Σ of MDs and a single MD $\varphi = \bigwedge_{i \in [1,k]} (R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$, if $\Sigma \models \varphi$, then $\Sigma \vdash_{\mathcal{I}} \varphi$. That is, if $\Sigma \models_m \varphi$, then φ can be derived from Σ by using the rules in $\mathcal{I}$.

The proof consists of two parts. (1) We first develop a *chase* procedure to compute the *closure* $(\Sigma, \varphi)^+$ of MDs. The closure is a set of triples of the form $(R_1[A], R_2[B], \rightleftharpoons)$ (or $(R[A], R'[B], \approx)$), where R, R' are in $\{R_1, R_2\}$, such that $\Sigma \models_m \mathsf{LHS}(\varphi) \rightarrow R_1[A] \rightleftharpoons R_2[B]$ (or $\Sigma \models_m \mathsf{LHS}(\varphi) \rightarrowtail R[A] \approx R'[B]$). (2) We then show that if $(R_1[Z_1[j]], R_2[Z_2[j]], \rightleftharpoons)$ is in $(\Sigma, \varphi)^+$ for all $j \in [1, |Z_1|]$, then $\Sigma \vdash_{\mathcal{I}} \mathsf{LHS}(\varphi) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$. From these it readily follows that $\mathcal{I}$ is complete.

(1) Chase. In the first part of the proof, we start with the chase process for MDs, by extending its counterpart for traditional dependencies (see, *e.g.,* [AHV95]). We then show that the chase process captures MD deduction.

To simplify the exposition we use the notations below:

- We use $(\Sigma, \varphi)^+ \models_m (R[A], R'[B], \mathsf{op})$ to denote that $(R[A], R'[B], \mathsf{op})$ is in $(\Sigma, \varphi)^+$, where op is either the matching operator $\rightleftharpoons$ or a similarity operator in Θ;
- Given MD $\phi = \bigwedge_{j \in [1,m]} (R_1[U_1[j]] \approx_j R_2[U_2[j]]) \rightarrow R_1[V_1] \rightleftharpoons R_2[V_2]$, we say $(\Sigma, \varphi)^+ \models_m \mathsf{LHS}(\phi)$ if and only if for each $j \in [1, m]$,
  1. $(\Sigma, \varphi)^+ \models_m (R_1[U_1[j]], R_2[U_2[j]], \rightleftharpoons)$ if $\approx_j$ is $=$, and
  2. $(\Sigma, \varphi)^+ \models_m (R_1[U_1[j]], R_2[U_2[j]], \approx_j)$ otherwise.

The chase process consists of seven steps as follows.

Step 1. Arrange the relations R_1 and R_2 in the MDs of Σ such that they are in the same order as in the MD φ.

Step 2. Initialize $(\Sigma, \varphi)^+$ with an *empty* set. For each pair $(R_1[X_1[i]], R_2[X_2[i]])$ ($i \in$

$[1,k]$) in the LHS of φ, let

- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[X_1[i]], R_2[X_2[i]], \rightleftharpoons)\}$ if $\approx_i$ is $=$; and
- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[X_1[i]], R_2[X_2[i]], \approx_i)\}$ otherwise.

Step 3. For each MD $\phi = \bigwedge_{j\in[1,m]}(R_1[U_1[j] \approx_j R_2[U_2[j]]) \rightarrow R_1[V_1] \rightleftharpoons R_2[V_2]$ in Σ, if $(\Sigma,\varphi)^+ \models_m \mathsf{LHS}(\phi)$, then let $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[V_1[j]], R_2[V_2[j]], \rightleftharpoons)\}$ for each $j \in [1,|V_1|]$.

Step 4. (a) If $(\Sigma,\varphi)^+ \models_m (R_1[E_1], R_2[D],$ op) and $(\Sigma,\varphi)^+ \models_m (R_1[E_2], R_2[D], \rightleftharpoons)$, then

- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_1], R_1[E_2], =)\}$, if op is $\rightleftharpoons$; and
- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_1], R_1[E_2],$op $)\}$ otherwise.

(b) similarly, if $(\Sigma,\varphi)^+ \models_m (R_1[C], R_2[F_1],$ op) and $(\Sigma,\varphi)^+ \models_m (R_1[C], R_2[F_2], \rightleftharpoons)$, then

- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_2[F_1], R_2[F_2], =)\}$ if op is $\rightleftharpoons$; and
- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_2[F_1], R_2[F_2],$op $)\}$ otherwise.

Step 5. (a) If $(\Sigma,\varphi)^+ \models_m (R_1[E_1], R_2[F_1], \rightleftharpoons)$ and $(\Sigma,\varphi)^+ \models_m (R_1[E_1], R_1[E_2], \approx)$, then

- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_2], R_2[F_1], \rightleftharpoons)\}$ if $\approx$ is $=$; and
- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_2], R_2[F_1], \approx)\}$ otherwise.

(b) If $(\Sigma,\varphi)^+ \models_m (R_1[E_1], R_2[F_1], \rightleftharpoons)$ and $(\Sigma,\varphi)^+ \models_m (R_2[F_1], R_2[F_2], \approx)$, then

- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_1], R_2[F_2], \rightleftharpoons)\}$ if $\approx$ is $=$; and
- $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_1], R_2[F_2], \approx)\}$ otherwise.

Step 6. (a) If $(\Sigma,\varphi)^+ \models_m (R_1[E_1], R_2[F_1], \approx)$ such that $\approx$ is not $=$ and $(\Sigma,\varphi)^+ \models_m (R_1[E_1], R_1[E_2], =)$, then let $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_2], R_2[F_1], \approx)\}$.

(b) If $(\Sigma,\varphi)^+ \models_m (R_1[E_1], R_2[F_1], \approx)$ such that $\approx$ is not $=$ and $(\Sigma,\varphi)^+ \models_m (R_2[F_1], R_2[F_2], =)$, then let $(\Sigma,\varphi)^+ = (\Sigma,\varphi)^+ \cup \{(R_1[E_1], R_2[F_2],$op $)\}$.

Step 7. Repeat steps 3, 4, 5 and 6 until no further changes can be made to the *closure* $(\Sigma,\varphi)^+$.

Termination. Given a set $\Sigma \cup \{\varphi\}$ of MDs, the chase process given above always terminates. Indeed, it stops after making at most $(|\Theta|+1) * h^2$ changes to $(\Sigma,\varphi)^+$ because of the following. First, the number h of attributes appearing in $\Sigma \cup \{\varphi\}$ is bounded by the total number of attributes in relations R_1 and R_2. Thus there are in total h^2 attribute pairs. Second, there are at most $|\Theta| + 1$ operators. Third, the number of elements in the closure $(\Sigma,\varphi)^+$ is bounded by $(|\Theta|+1) * h^2$.

Chase Property. We now show that if $\Sigma \models_m \varphi$, then for each $j \in [1,|Z_1|]$, $(\Sigma,\varphi)^+ \models_m (R_1[Z_1[j]], R_2[Z_2[j]], \rightleftharpoons)$, denoted by $(\Sigma,\varphi)^+ \models_m R_1[Z_1] \rightleftharpoons R_2[Z_2]$.

We prove this by contradictions. Assume that $\Sigma \models_m \varphi$, but $(\Sigma,\varphi)^+ \not\models_m R_1[Z_1] \rightleftharpoons$

$R_2[Z_2]$. That is, there exists $j \in [1, |Z_1|]$ such that $(\Sigma, \varphi)^+ \not\models_m R_1[Z_1[j]] \rightleftharpoons R_2[Z_2[j]]$. Let $Z_1[j]$ be A and $Z_2[j]$ be B. We construct a pair (D, D') of instances based on $(\Sigma, \varphi)^+$ such that $(D, D') \models \Sigma$ but $(D, D') \not\models \varphi$. This contradicts the assumption that $\Sigma \models_m \varphi$.

We construct such (D, D') based on a *small model property* of MDs. That is, if $\Sigma \not\models_m \varphi$, then there exists a two-tuple instance $D = (I_1, I_2)$ of (R_1, R_2), where I_1 (resp. I_2) consists of a single tuple t_1 (resp. t_2), such that there exists a stable instance D', $(D, D') \models \Sigma$, but $(D, D') \not\models \varphi$. This property is easy to verify. In light of this, we shall construct D and D' consisting of two tuples only.

We now give the construction of (D, D'). We first group attributes and assign a unique constant to each group of attributes. We then build (D, D') from these attribute groups.

(1) Grouping attributes. We group the attributes by defining an equivalence relation. Let $\mathsf{attr}(R)$ denote the set of attributes in a relation schema R. For any attributes A, B in $\mathsf{attr}(R_1) \cup \mathsf{attr}(R_2)$, we say that A and B are *equivalent* if either $(A, B, =)$ or $(A, B, \rightleftharpoons)$ is in the closure $(\Sigma, \varphi)^+$.

We compute the equivalence classes as follows.

- For each attribute A in $\mathsf{attr}(R_1) \cup \mathsf{attr}(R_2)$, create an equivalent class consisting of itself only. We use EQ to represent all those equivalent classes, and eq_A to represent the equivalent class that attribute A belongs to.
- For any attributes A and B in $\mathsf{attr}(R_1) \cup \mathsf{attr}(R_2)$, do the following.
  - If $(A, B, =)$ or $(A, B, \rightleftharpoons)$ is in the closure $(\Sigma, \varphi)^+$, then merge eq_A and eq_B. That is, we let $\mathsf{EQ} = (\mathsf{EQ} \setminus \{eq_A, eq_B\}) \cup \{eq_{AB}\}$, where $eq_{AB} = eq_A \cup eq_B$.
  - If $(A, B, \approx)$ is in the closure $(\Sigma, \varphi)^+$, where $\approx$ is not equality, then mark $eq_A \approx eq_B$.

For each equivalent class $eq \in \mathsf{EQ}$, we assign a constant c, denoted by $eq.c$, such that for two distinct equivalent classes eq_1 and eq_2 in EQ, (a) $eq_1.c \neq eq_2.c$, (b) $eq_1.c \approx eq_2.c$ if $eq_1 \approx eq_2$, and (c) $eq_1.c \not\approx eq_2.c$ if $eq_1 \not\approx eq_2$. It is possible to find such constants since we consider dense similarity operators (see Section 6.3.1).

(2) Instance construction. Based on the equivalence classes, we construct the pair (D, D') of instances as follows.

- Let t_1 be a tuple of relation R_1 such that $t_1[A] = eq_A.c$ for each attribute $A \in \mathsf{attr}(R_1)$.
- Let t_2 be a tuple of relation R_2 such that $t_2[B] = eq_B.c$ for each attribute $B \in \mathsf{attr}(R_2)$.

- Let I_1 (resp. I_2) be an instance of relation R_1 (resp. R_2) consisting of the tuple t_1 (resp. t_2) only.
- Finally, let $D = (I_1, I_2)$, and $D' = D$.

(3) Verification. We show that (D, D) constructed above are indeed a counterexample. That is, if $\Sigma \models_m \varphi$ but $(\Sigma, \varphi)^+ \not\models_m R_1[A] \rightleftharpoons R_2[B]$, then $(D, D) \models \Sigma$, but $(D, D) \not\models \varphi$.

We first show that $(D, D) \not\models \mathsf{LHS}(\varphi) \rightarrow R_1[A] \rightleftharpoons R_2[B]$, where $\mathsf{LHS}(\varphi)$ is $\bigwedge_{i\in[1,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]])$. Indeed, $t_1[A] \neq t_2[B]$ since eq_A and eq_B are distinct equivalence classes, by $(\Sigma, \varphi)^+ \not\models_m R_1[A] \rightleftharpoons R_2[B]$. Furthermore, for each $i \in [1, |X_1|]$, $eq_{X_1[j]} \approx_i eq_{X_2[j]}$ and hence, $t_1[X_1[j]] \approx_i t_2[X_2[j]]$ by the construction of D. As a result, $(D, D) \not\models \mathsf{LHS}(\varphi) \rightarrow R_1[A] \rightleftharpoons R_2[B]$. Hence $(D, D) \not\models \varphi$.

We then show that $(D, D) \models \Sigma$. Assume by contradiction that there exists ϕ in Σ such that $(D, D) \not\models \phi$, where ϕ is $\bigwedge_{i\in[1,k]}(R_1[Z_1[i]] \approx_i R_2[Z_2[i]]) \rightarrow R_1[W_1] \rightleftharpoons R_2[W_2]$. That is, (t_1, t_2) match $\mathsf{LHS}(\phi)$ but $t_1[W_1] \neq t_2[W_2]$. By the construction of D, if (t_1, t_2) match $\mathsf{LHS}(\phi)$, then for each $i \in [1, |Z_1|]$, $(\Sigma, \varphi)^+ \models_m R_1[X_1[i]] \rightleftharpoons R_2[X_2[i]]$. Then by the chase process given above, for each $j \in [1, |W_1|]$, $(W_1[j], W_2[j], \rightleftharpoons)$ would have been included in $(\Sigma, \varphi)^+$, or in other words, $(\Sigma, \varphi)^+ \models_m R_1[W_1] \rightleftharpoons R_2[W_2]$. Again by the construction of D, we would have had that $t_1[W_1] = t_2[W_2]$, which contradicts the assumption. Hence $(D, D) \models \Sigma$.

Therefore, if $\Sigma \models_m \varphi$ then $(\Sigma, \varphi)^+ \models_m \mathsf{RHS}(\varphi)$.

(2) Simulation of the chase process. We next give the second part of the proof, by showing the following. (a) If $(\Sigma, \varphi)^+ \models_m (R_1[A], R_2[B], \rightleftharpoons)$, then $\Sigma \vdash_{\mathcal{I}} \mathsf{LHS}(\varphi) \rightarrow R_1[A] \rightleftharpoons R_2[B]$. (b) If $(\Sigma, \varphi)^+ \models_m (R[A], R'[B], \approx)$, then $\Sigma \vdash_{\mathcal{I}} \mathsf{LHS}(\varphi) \rightarrowtail R[A] \approx R'[B]$ where R, R' are relations in $\{R_1, R_2\}$ and $\approx$ is not $=$ when $R \neq R'$. If this holds, then we can conclude that if $(\Sigma, \varphi)^+ \models_m R_1[Z_1] \rightleftharpoons R_2[Z_2]$, then $\Sigma \vdash_{\mathcal{I}} \varphi$.

It suffices to show that each step of the chase process is an application of certain inference rules in $\mathcal{I}$. For if it holds, then the computation of $(\Sigma, \varphi)^+$ corresponds to a proof using rules in $\mathcal{I}$. In other words, the chase process to compute $(\Sigma, \varphi)^+ \models_m R_1[Z_1] \rightleftharpoons R_2[Z_2]$ yields a proof of $\Sigma \vdash_{\mathcal{I}} \varphi$.

Step 1. This corresponds to an application of MD_1.

Step 2. It is justified by an application of MD_2.

Step 3. This corresponds to applications of MD_4 and MD_5. More specifically, since $(\Sigma, \varphi)^+ \models_m \mathsf{LHS}(\phi)$, for each $j \in [1, m]$ we can derive the following: (a) $\Sigma \vdash_{\mathcal{I}} \phi_1$ if $\approx_j$ is $=$, where ϕ_1 is $\mathsf{LHS}(\phi) \rightarrow R_1[U_1[j]] \rightleftharpoons R_2[U_2[j]]$; and (b) $\Sigma \vdash_{\mathcal{I}} \phi_2$ otherwise, where ϕ_2 is either $\mathsf{LHS}(\phi) \rightarrow R_1[U_1[j]] \rightleftharpoons R_2[U_2[j]]$ or $\mathsf{LHS}(\phi) \rightarrowtail R_1[U_1[j]] \approx_j R_2[U_2[j]]$. Hence

by applying MD_4 to ϕ_1 and ϕ_2, we have that $\Sigma \vdash_{\mathcal{I}} \mathsf{LHS}(\varphi) \rightarrow \mathsf{RHS}(\phi)$. Note that when $\approx_j$ is not $=$ and when only $(R_1[U_1[j]], R_2[U_2[j]], \rightleftharpoons)$ is in $(\Sigma, \varphi)^+$, MD_5 needs to be applied to ϕ first in order to replace $\approx_j$ with $=$.

Step 4. This is justified by applications of MD_2, MD_3, MD_4, MD_6 and MD_7. We verify this for case (a) as follows; the proof for case (b) is similar. For case (a) we further distinguish two cases, depending on whether op is $\rightleftharpoons$ or not.

(1) When op is $\rightleftharpoons$. Since $(\Sigma, \varphi)^+ \models_m (R_1[E_1], R_2[D], \rightleftharpoons)$ and $(\Sigma, \varphi)^+ \models_m (R_1[E_2], R_2[D], \rightleftharpoons)$, we have that $\Sigma \vdash_{\mathcal{I}} \phi_1$ and $\Sigma \vdash_{\mathcal{I}} \phi_2$, where $\phi_1 = \mathsf{LHS}(\varphi) \rightarrow R_1[E_1] \rightleftharpoons R_2[D]$ and $\phi_2 = \mathsf{LHS}(\varphi) \rightarrow R_1[E_2] \rightleftharpoons R_2[D]$. We conduct deduction analysis based on $\mathcal{I}$ as follows.

- By applying rule MD_3 to ϕ_1, we deduce that $\phi_3 = \mathsf{LHS}(\varphi) \wedge (R_1[E_1] = R_2[D]) \rightarrow R_1[E_1E_2] \rightleftharpoons R_2[FF]$.
- By applying rules MD_2 and MD_4 to ϕ_1 and ϕ_3, we have that $\phi_4 = \mathsf{LHS}(\varphi) \rightarrow R_1[E_1E_2] \rightleftharpoons R_2[FF]$.
- Finally, by applying MD_6 to ϕ_4, we can deduce that $\Sigma \vdash_{\mathcal{I}} \mathsf{LHS}(\varphi) \rightarrowtail R_1[E_1] = R_1[E_2]$.

(2) When op is a non-equality similarity operator in Θ. We can derive that $\Sigma \vdash_{\mathcal{I}} \mathsf{LHS}(\varphi) \rightarrowtail R_1[E_1]$ op $R_1[E_2]$ by using a similar argument, except that we use MD_7 here instead of MD_6.

Step 5. This is justified by an application of MD_8. We prove case (a) of the step below; the proof for case (b) is similar.

Since $(\Sigma, \varphi)^+ \models_m (R_1[E_1], R_2[F_1], \rightleftharpoons)$ and $(\Sigma, \varphi)^+ \models_m (R_1[E_1], R_1[E_2], \approx)$, we have that $\Sigma \vdash_{\mathcal{I}} \phi_1$ and $\Sigma \vdash_{\mathcal{I}} \phi_2$, where $\phi_1 = \mathsf{LHS}(\varphi) \rightarrow R_1[E_1] \rightleftharpoons R_2[D]$ and $\phi_2 = \mathsf{LHS}(\varphi) \rightarrowtail R_1[E_1] \approx R_1[E_2]$. By applying MD_8 to ϕ_1 and ϕ_2, we can deduce $\phi_3 = \mathsf{LHS}(\varphi) \rightarrow (R_1[E_2] \rightleftharpoons R_2[F_1])$ if $\approx$ is $=$, and $\phi_4 = \mathsf{LHS}(\varphi) \rightarrowtail (R_1[E_2] \approx R_2[F_1])$ otherwise.

Step 6. This step is justified by an application of MD_9. Again we only prove case (a) of the step; the proof for case (b) is similar.

From $(\Sigma, \varphi)^+ \models_m (R_1[E_1], R_2[F_1], \approx)$ and $(\Sigma, \varphi)^+ \models_m (R_1[E_1], R_1[E_2], =)$, it follows that $\Sigma \vdash_{\mathcal{I}} \phi_1$ and $\Sigma \vdash_{\mathcal{I}} \phi_2$, where $\phi_1 = \mathsf{LHS}(\varphi) \rightarrow R_1[E_1] \approx R_2[F_1]$ and $\phi_2 = \mathsf{LHS}(\varphi) \rightarrowtail R_1[E_1] = R_1[E_2]$. By applying MD_9 to ϕ_1 and ϕ_2, we can deduce that $\phi_3 = \mathsf{LHS}(\varphi) \rightarrow (R_1[E_2] \approx R_2[F_1])$.

Putting the two parts of proofs together, we have shown the following: (a) if $\Sigma \models_m \varphi$, then $(\Sigma, \varphi)^+ \models_m R_1[Z_1] \rightleftharpoons R_2[Z_2]$, and (b) if $(\Sigma, \varphi)^+ \models_m R_1[Z_1] \rightleftharpoons R_2[Z_2]$, then $\Sigma \vdash_{\mathcal{I}} \varphi$. Hence we can conclude that if $\Sigma \models_m \varphi$, then $\Sigma \vdash_{\mathcal{I}} \varphi$, *i.e.,* rules MD_1–MD_9 are

complete for MD deduction. □

6.4 An Algorithm for Deduction Analysis

We next focus on the deduction problem for matching dependencies. The main result of this section is the following:

Theorem 6.4.1: *There exists an algorithm that, given as input a set Σ of* MD*s and another* MD φ *over schemas (R_1, R_2), determines whether or not $\Sigma \models_m \varphi$ in $O(n^2 + h^3)$ time, where n is the size of Σ and φ, and h is the total number of distinct attributes appearing in Σ or φ.* □

The algorithm is in quadratic-time in the size of the input when (R_1, R_2) are fixed. Indeed, h is no larger than the arity of (R_1, R_2) (the total number of attributes in (R_1, R_2)) and is typically much smaller than n. It should be remarked that the deduction analysis of MDs is carried out at compile time on MDs, which are *much smaller* than data relations on which record matching is performed.

Compared to the $O(n)$-time complexity of FD implication, Theorem 6.4.1 tells us that although the expressive power of MDs is not for free, it does not come at too big a price.

Below we prove Theorem 6.4.1 by first developing the algorithm and then verifying the correctness of the algorithm. In the next section we shall leverage the algorithm when computing a set of quality RCKs.

Overview. To simplify the discussion we consider *w.l.o.g.* a normal form of MDs. We consider MDs ϕ of the form:

$$\bigwedge_{j\in[1,m]}(R_1[U_1[j]] \approx_j R_2[U_2[j]]) \rightarrow R_1[A] \rightleftharpoons R_2[B],$$

i.e., $\mathsf{RHS}(\phi)$ is a single pair of attributes in (R_1, R_2). This does not lose generality as an MD ψ of the general form, *i.e.,* when $\mathsf{RHS}(\psi)$ is (Z_1, Z_2), is equivalent to a set of MDs in the normal form, one for each pair of attributes in (Z_1, Z_2), by rules MD_2, MD_3 and MD_4 in the inference system $\mathcal{I}$.

In particular, we assume that the input MD φ is:

$$\varphi = \bigwedge_{i\in[1,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[E_1] \rightleftharpoons R_2[E_2].$$

The algorithm, referred to as $\mathsf{MDClosure}$, takes MDs Σ and φ as input, and computes the *closure* of Σ and $\mathsf{LHS}(\varphi)$. The closure is the set of all pairs $(R_1[A], R_2[B])$ such that $\Sigma \models_m \mathsf{LHS}(\varphi) \rightarrow R_1[A] \rightleftharpoons R_2[B]$, similar to the one used in the proof of Lemma 6.3.3. Thus one can conclude that $\Sigma \models_m \varphi$ if and only if $(R_1[E_1], R_2[E_2])$ is in the closure.

The closure of Σ and φ is stored in an $h \times h \times p$ array M. The first two dimensions are indexed by distinct attributes appearing in Σ or φ, and the last one by distinct similarity operators in Σ or φ (including $=$). Note that $p \leq |\Theta|$, where the set Θ of similarity metrics is *fixed*. In practice, p is a *constant*: in any application domain only a small set of *predefined* similarity metrics is used.

The algorithm computes M based on Σ and $\mathsf{LHS}(\varphi)$ such that for relation schemas R, R' and for similarity operator $\approx$, $M(R[A], R'[B], \approx) = 1$ iff $\Sigma \models_m \mathsf{LHS}(\varphi) \rightarrow R[A] \approx R'[B]$. Here we use weak MDs to express intermediate results during the computation, *i.e.,* we allow R and R' to be the same relation (either R_1 or R_2), and $\approx$ to appear in the RHS of MDs. As shown by rules MD_6, MD_7, MD_8 and MD_9 in the inference system $\mathcal{I}$, this may happen due to the interaction between the matching operator and similarity operators.

Putting these together, algorithm MDClosure takes Σ and φ as input, computes the closure of Σ and $\mathsf{LHS}(\varphi)$ using M, and concludes that $\Sigma \models_m \varphi$ iff $M(R_1[E_1], R_2[E_2], =)$ is 1. By the inference system $\mathcal{I}$, we can set $M(R_1[E_1], R_2[E_2], =)$ = 1 iff $R_1[E_1] \rightleftharpoons R_2[E_2]$ is deduced from Σ and $\mathsf{LHS}(\varphi)$.

Algorithm. Algorithm MDClosure is given in Fig. 6.5. While the algorithm is along the same lines as its counterpart for FD implication [AHV95], it is far more involved. Indeed, MD deduction has to deal with intriguing interactions between the matching operator and similarity operators. Below we first present procedures for handling the interactions.

Procedure AssignVal. As shown in Fig. 6.5, this procedure takes a similar pair $R[A] \approx R'[B]$ as input. It checks whether or not $M(R[A], R'[B], \approx)$ or $M(R[A], R'[B], =)$ is already set to 1 (line 1). If not, it sets both $M(R[A], R'[B], \approx)$ and its symmetric entry $M(R'[B], R[A], \approx)$ to 1, and returns true (lines 2–3). Otherwise it returns false (line 4).

Observe that if $M(R[A], R'[B], =)$ is 1, then no change is needed, since from $R[A] = R'[B]$ it follows that $R[A] \approx R'[B]$. Indeed, the generic axioms for similarity operators tell us that each similarity relation $\approx$ subsumes $=$.

Procedures Propagate *and* Infer. When $M(R[A], R'[B], \approx)$ is changed to 1, the change may have to be propagated to other M entries. Indeed, by the generic axioms for similarity operators, we have the following:
(1) for each $R[E] = R[A]$ (resp. $R'[E] = R[A]$), it follows that $R[E] \approx R'[B]$ (resp. $R'[E] \approx R'[B]$). Hence entries $M(R[E], R'[B], \approx)$ (resp. $M(R'[E], R'[B], \approx)$) should also be set to 1; similarly for $R[E] = R'[B]$.

Algorithm MDClosure

Input: A set Σ of MDs and another MD φ, where $\mathsf{LHS}(\varphi) = \bigwedge_{i\in[1,k]}(R_1[X_1[i]]\approx_i R_2[X_2[i]])$.

Output: The closure of Σ and $\mathsf{LHS}(\varphi)$, stored in array M.

1. All entries of M are initialized to 0;
2. **for** each $i \in [1,k]$ **do**
3. **if** AssignVal $(M, R_1[X_1[i]], R_2[X_2[i]], \approx_i)$ **then**
4. Propagate $(M, R_1[X_1[i]], R_2[X_2[i]], \approx_i)$;
5. **repeat** until no further changes
6. **for** each MD ϕ in Σ **do**

/* $\phi = \bigwedge_{j\in[1,m]}(R_1[U_1[j]]\approx_j R_2[U_2[j]]) \rightarrow R_1[A] \rightleftharpoons R_2[B]$ */

7. **if** there is $d \in [1,m]$ such that $M(R_1[U_1[d]], R_2[U_2[d]], =) = 0$
and $M(R_1[U_1[d]], R_2[U_2[d]], \approx_d) = 0$ $(1 \le d \le m)$ **then**
8. **continue** ;
9. **else** $\{\Sigma := \Sigma \setminus \{\phi\}$;
10. **if** AssignVal $(M, R_1[A], R_2[B], =)$ **then**
11. Propagate $(M, R_1[A], R_2[B], =)$;$\}$
12. **return** M.

Procedure AssignVal $(M, R[A], R'[B], \approx)$

Input: Array M with new similar pair $R[A] \approx R'[B]$.

Output: Update M, return true if M is updated and false otherwise.

1. **if** $M(R[A], R'[B], =) = 0$ **and** $M(R[A], R'[B], \approx) = 0$ **then**
2. $M(R[A], R'[B], \approx) := 1$; $M(R'[B], R[A], \approx) := 1$;
3. **return** true;
4. **else return** false;

Figure 6.5: Algorithm MDClosure

(2) If $\approx$ is $=$, then for each $R[E] \approx_d R[A]$ (resp. $R'[E] \approx_d R[A]$), we have that $R[E] \approx_d R'[B]$ (resp. $R'[E] \approx_d R'[B]$); and hence, $M(R[E], R'[B], \approx_d)$ (resp. $M(R'[E], R'[B], \approx_d)$) has to be set to 1.

In turn these changes may trigger new changes to M, and so on. It is to handle this that procedures Propagate and Infer are used, which recursively propagate the changes.

These procedures are given in Fig. 6.6. They use a queue Q to keep track of and process the changes: changes are pushed into Q whenever they are encountered, and are popped off from Q and processed one by one until Q is empty.

More specifically, procedure Propagate takes a newly deduced similar pair $R[A] \approx$

Procedure Propagate $(M, R_1[A], R_2[B], \approx)$

Input: Array M with updated similar pair $R_1[A] \approx R_2[B]$.

Output: Updated M to include similarity change propagation.

1. Q.push$(R_1[A], R_2[B], \approx)$;
2. **while** (Q is not empty) **do**
3. $(R[E], R'[E'], \approx_d)$:= Q.pop();
4. **case** (R, R') of
5. (1) $R = R_1$ **and** $R' = R_2$
6. Infer$(Q, M, R_2[E'], R_1[E], R_1, \approx_d)$; Infer$(Q, M, R_1[E], R_2[E'], R_2, \approx_d)$;
7. (2) $R = R' = R_1$
8. Infer$(Q, M, R_1[E], R_1[E'], R_2, \approx_d)$; Infer$(Q, M, R_1[E'], R_1[E], R_2, \approx_d)$;
9. (3) $R = R' = R_2$
10. Infer$(Q, M, R_2[E], R_2[E'], R_1, \approx_d)$; Infer$(Q, M, R_2[E'], R_2[E], R_1, \approx_d)$;

Procedure Infer$(Q, M, R[A], R'[B], R'', \approx)$

Input: Queue Q, array M, newly updated similar pair
$R[A] \approx R'[B]$, and relation name R''.

Output: New similar pairs stored in Q and updated M.

1. **for** each attribute E of R'' **do**
2. **if** $M(R[A], R''[E], =) = 1$ **and** AssignVal $(M, R'[B], R''[E], \approx)$ **then**
3. Q.push$(R'[B], R''[E], \approx)$;}
4. **if** $\approx$ is $=$ **then**
5. **for** each similarity operator $\approx_d$ $(1 \leq d \leq p)$ **do**
6. **if** $M(R[A], R''[E], \approx_d) = 1$ **and** AssignVal$(M, R'[B], R''[E], \approx_d)$ **then**
7. Q.push$(R'[B], R''[E], \approx_d)$;

Figure 6.6: Procedures Propagate and Infer

$R'[B]$ as input, and updates M accordingly. It first pushes the pair into Q (line 1). Then for each entry $R[E] \approx R'[E']$ in Q (line 3), three different cases are considered, depending on whether (R, R') are (R_1, R_2) (lines 5–6), (R_1, R_1) (lines 7–8) or (R_2, R_2) (lines 9–10). In each of these cases, procedure Infer is invoked, which modifies M entries based on the generic axioms for similarity operators given in Section 6.2. The process proceeds until Q becomes empty (line 2).

Procedure Infer takes as input the queue Q, array M, a new similar pair $R[A] \approx R'[B]$, and relation R'', where R, R', R'' are either R_1 or R_2. It infers other similar pairs, pushes them into Q, and invokes procedure AssignVal to update corresponding M entries. It

handles two cases, namely, the cases (1) and (2) mentioned above (lines 2–3 and 4–7, respectively). The new pairs pushed into Q are processed by procedure Propagate, as described above.

Algorithm MDClosure. We are now ready to illustrate the main driver of the algorithm (Fig. 6.5), which works as follows. It first sets all entries of array M to 0 (line 1). Then for each pair $R_1[X_1[i]] \approx_i R_2[X_2[i]]$ in $\mathsf{LHS}(\varphi)$, it stores the similar pair in M (lines 2–4). After these initialization steps, the algorithm inspects each MD ϕ in Σ one by one (lines 6–11). It checks whether $\mathsf{LHS}(\phi)$ is matched (line 7), and if so, it invokes procedures AssignVal and Propagate to update M based on $\mathsf{RHS}(\phi)$, and propagate the changes (line 10–11). The inspection of $\mathsf{LHS}(\phi)$ uses a property mentioned earlier: if $M(R_1[U_1], R_2[U_2], =) = 1$, then $R_1[U_1] \approx_d R_2[U_2]$ for any similarity metric $\approx_d$ (line 7). Once an MD is applied, it will not be inspected again (line 9). The process proceeds until no more changes can be made to array M (line 5). Finally, the algorithm returns M (line 12).

Example 6.4.1: Recall Σ_c and rck_4 from Example 6.3.5. We show how rck_4 is deduced from Σ_c by MDClosure. We use the table below to keep track of the changes to array M after step 4 of the algorithm, when MDs in Σ_c are applied. We use c and b to denote relations credit and billing, respectively.

step	new updates to M
step 4	M(c[email], b[email], =) = M(b[email], c[email], =) = 1 M(c[tel], b[phn], =) = M(b[phn], c[tel], =) = 1
φ_2	M(c[addr], b[post], =) = M(b[post], c[addr], =) = 1
φ_3	M(c[FN], b[FN], =) = M(b[FN], c[FN], =) = 1 M(c[LN], b[LN], =) = M(b[LN], c[LN], =) = 1
φ_1	M(c[Y_c], b[Y_b], =) = M(b[Y_b], c[Y_c],=) = 1

After step 4, M is initialized with $c[\mathsf{email}] = b[\mathsf{email}]$ and $c[\mathsf{tel}] = b[\mathsf{phn}]$, as given by $\mathsf{LHS}(\mathsf{rck}_4)$. Now both $\mathsf{LHS}(\varphi_2)$ and $\mathsf{LHS}(\varphi_3)$ are matched, and thus M is updated with $c[\mathsf{addr}] \rightleftharpoons b[\mathsf{post}]$ (as indicated by $M(c[\mathsf{addr}], b[\mathsf{post}], =)$), $c[\mathrm{FN}] \rightleftharpoons b[\mathrm{FN}]$ and $c[\mathrm{LN}] \rightleftharpoons b[\mathrm{LN}]$. As a result of the changes, $\mathsf{LHS}(\varphi_1)$ is matched, and $M(c[Y_c], b[Y_b], =)$ is set to 1. After that, no more changes can be made to array M. Since $M(c[Y_c], b[Y_b], =) = 1$, we conclude that $\Sigma \models_m \mathsf{rck}_4$.

As another example, we show how MDClosure deduces ψ from $\{\psi_1, \psi_2, \psi_3\}$, where ψ, ψ_1, ψ_2 and ψ_3 are:

$$\psi \ = R_2[A_2] = R_1[A_1] \rightarrow R_2[E_2] \rightleftharpoons R_1[B_1],$$

$$\psi_1 = R_1[A_1] \approx R_2[A_2] \rightarrow R_1[B_1] \rightleftharpoons R_2[B_2],$$
$$\psi_2 = R_1[A_1] \approx R_2[A_2] \rightarrow R_1[E_1] \rightleftharpoons R_2[B_2],$$
$$\psi_3 = R_1[A_1] \approx R_2[A_2] \rightarrow R_1[E_1] \rightleftharpoons R_2[E_2].$$

We use the table below to show how MDClosure computes array M. After step 4, M stores $R_1[A_1] = R_2[A_2]$ to reflect $\mathsf{LHS}(\psi)$. Then $\mathsf{LHS}(\psi_1)$, $\mathsf{LHS}(\psi_2)$ and $\mathsf{LHS}(\psi_3)$ are matched. Applying ψ_1 first, $R_1[B_1] \rightleftharpoons R_2[B_2]$ is added to M. Now apply ψ_2, and M is updated with $R_1[E_1] \rightleftharpoons R_2[B_2]$. Here procedure $\mathsf{Infer}(Q, M, R_2[B_2], R_1[E_1], R_1)$ deduces a new pair $R_1[B_1] \rightleftharpoons R_1[E_1]$ from $R_1[B_1] \rightleftharpoons R_2[B_2]$ and $R_1[E_1] \rightleftharpoons R_1[B_2]$, and AssignVal is called to update M accordingly. Similarly, when ψ_3 is applied, $R_1[E_1] \rightleftharpoons R_2[E_2]$ is added to M. When Propagate and Infer are invoked, they further infer $R_2[E_2] \rightleftharpoons R_2[B_2]$ and $R_1[B_1] \rightleftharpoons R_2[E_2]$. Accordingly M is updated to keep track of these changes.

step	**new updates**
step 4	$M(R_1[A_1], R_2[A_2], =) = M(R_2[A_2], R_1[A_1], =) = 1$
ψ_1	$M(R_1[B_1], R_2[B_2], =) = M(R_2[B_2], R_1[B_1], =) = 1$
ψ_2	$M(R_1[E_1], R_2[B_2], =) = M(R_2[B_2], R_1[E_1], =) = 1$
	$M(R_1[E_1], R_1[B_1], =) = M(R_1[B_1], R_1[E_1], =) = 1$
ψ_3	$M(R_1[E_1], R_2[E_2], =) = M(R_2[E_2], R_1[E_1], =) = 1$
	$M(R_2[E_2], R_2[B_2], =) = M(R_2[B_2], R_2[E_2], =) = 1$
	$M(R_1[B_1], R_2[E_2], =) = M(R_2[E_2], R_1[B_1], =) = 1$

After ψ_3 is applied, M can no longer be changed. Hence $\{\psi_1, \psi_2, \psi_3\} \models_m \psi$, by $M(R_2[E_2], R_1[B_1], =) = 1$. □

Complexity analysis. MDClosure executes the **repeat** loop at most n times, since in each iteration it calls procedure Propagate, which applies at least one MD in Σ. That is, Propagate can be called at most n times *in total*. Each iteration searches at most all MDs in Σ. For the k-th call of Propagate ($1 \leq k \leq n$), let L_k be the number of while-loops it executes. For each loop, it takes at most $O(h)$ time since procedure Infer is in $O(h)$ time. Hence *the total cost* of updating array M is in $O((L_1 + \ldots + L_n)h)$ time. Note that $(L_1 + \ldots + L_n)$ is the total number of changes made to array M, which is bounded by $O(h^2)$. Putting these together, algorithm MDClosure is in $O(n^2 + h^3)$ time. As remarked earlier, h is usually much smaller than n, and is a constant when (R_1, R_2) are fixed. Hence the algorithm is in $O(n^2)$ time in practice. Furthermore, the algorithm can be improved by leveraging the index structures of [BB79, Mai83] for FD implication.

Finally we give a proof of Theorem 6.4.1.

Proof of Theorem 6.4.1. It suffices to show that for any Σ and φ as described above, $\Sigma \models_m \varphi$ if and only if Algorithm MDClosure sets $M(R_1[E_1], R_2[E_2], =) = 1$. For if this

holds, then by the complexity analysis given above, Algorithm MDClosure is precisely the algorithm we want.

Recall the notion of $(\Sigma,\varphi)^+ \models_m (R_1[E_1],R_2[E_2],\rightleftharpoons)$ from the proof of Lemma 6.3.3. It is already shown there that $\Sigma \models_m \varphi$ if and only if $(\Sigma,\varphi)^+ \models_m (R_1[E_1],R_2[E_2],\rightleftharpoons)$. Thus to verify the correctness of Algorithm MDClosure, it suffices to show that $(\Sigma,\varphi)^+ \models_m (R_1[E_1],R_2[E_2],\rightleftharpoons)$ if and only if MDClosure sets $M(R_1[E_1],R_2[E_2],=) = 1$.

First assume that $(\Sigma,\varphi)^+ \models_m (R_1[E_1],R_2[E_2],\rightleftharpoons)$. Then one can verify that MDClosure sets $M(R_1[E_1],R_2[E_2],=)$ to 1 by an induction on the steps when $(R_1[E_1],R_2[E_2],\rightleftharpoons)$ is included in $(\Sigma,\varphi)^+$. Indeed, there is a straightforward correspondence between the steps of chase process given in the proof of Lemma 6.3.3 and the steps of MDClosure. Based on the correspondence the induction can be readily conducted.

Conversely, assume that $M(R_1[E_1],R_2[E_2],=)$ is set to 1 by MDClosure. One can show that $(R_1[E_1],R_2[E_2],\rightleftharpoons)$ is included in $(\Sigma,\varphi)^+$ by induction on the steps when MDClosure sets $M(R_1[E_1],R_2[E_2],=)$ to 1. The induction is again based on the correspondence between the steps of MDClosure and the steps of the chase process. □

6.5 Computing Relative Candidate Keys

As remarked in Section 6.1, to improve match quality we often need to repeat blocking, windowing and matching processes multiple times, each using a different key [EIV07].

This gives rise to *the problem for computing* RCK*s*: given a set Σ of MDs, a pair of comparable lists (Y_1,Y_2), and a natural number m, it is to compute a set Γ of m quality RCKs relative to (Y_1,Y_2), deduced from Σ.

This problem is nontrivial. One question concerns what metrics we should use to select RCKs. Another question is how to find m quality RCKs using the metric. One might be tempted to first compute all RCKs from Σ, sort these keys based on the metric, and then select the top m keys. This is, however, beyond reach in practice: it is known that for a single relation, there are possibly exponentially many traditional candidate keys [LO78]. For RCKs, unfortunately, the exponential-time complexity remains intact.

In this section we first propose a model to assess the quality of RCKs. Based on the model, we then develop an efficient algorithm to infer m RCKs from Σ. As will be verified by our experimental study, even when Σ does not contain many MDs, the algorithm is still able to find a reasonable number of RCKs. In addition, in practice it

is rare to find exponentially many RCKs; indeed, the algorithm often finds the set of *all* quality RCKs when m is not very large.

Quality model. To construct the set Γ, we select RCKs based on the following criteria.

- The *diversity* of RCKs in Γ. We do not want those RCKs defined with pairs $(R_1[A], R_2[B])$ if the pairs appear frequently in RCKs that are already in Γ. That is, we want Γ to include diverse attributes so that if errors appear in some attributes, matches can still be found by comparing other attributes in the RCKs of Γ. To do this we maintain a counter $\mathsf{ct}(R_1[A], R_2[B])$ for each pair, and increase it by 1 whenever an RCK with the pair is added to Γ.
- Statistics. We consider the accuracy of each attribute pair $\mathsf{ac}(R_1[A], R_2[B])$, *i.e.,* the confidence placed by the user in the attributes, and average lengths $\mathsf{lt}(R_1[A], R_2[B])$ of the values of each attribute pair. Intuitively, the longer $\mathsf{lt}(R_1[A], R_2[B])$ is, the more likely errors occur in the attributes; and the greater $\mathsf{ac}(R_1[A], R_2[B])$ is, the more reliable $(R_1[A], R_2[B])$ are.

Putting these together, we define the *cost* of including attributes $(R_1[A], R_2[B])$ in an RCK as:

$$\mathsf{cost}(R_1[A], R_2[B]) = w_1 \cdot \mathsf{ct}(R_1[A], R_2[B]) + w_2 \cdot \mathsf{lt}(R_1[A], R_2[B]) + w_3 / \mathsf{ac}(R_1[A], R_2[B])$$

where w_1, w_2, w_3 are weights associated with these factors. Our algorithm selects RCKs with attributes of low cost or equivalently, high quality.

Overview. We focus on RCKs $(X_1, X_2, [\approx_1, \ldots, \approx_k])$ such that for each $i \in [1,k]$, $R_1[X_1[i]] \approx_i R_2[X_2[i]]$ appears in either Σ or in the default relative key $(Y_1, Y_2, [=, \ldots, =])$. The reason is twofold. First, we want to preserve attribute pairs specified by MDs in Σ, which are identified as attributes that are sensible to compare either by domain experts or by learning from sample data. Second, by focusing on such RCKs one does not have to worry about weak MDs in the deduction process, and hence it reduces the computational cost. We refer to such RCKs as *normal* RCKs.

We provide an algorithm for computing RCKs, referred to as findRCKs. Given Σ, (Y_1, Y_2) and m as input, it returns a set Γ of at most m RCKs relative to (Y_1, Y_2) that are deduced from Σ. The algorithm selects RCKs defined with low-cost attribute pairs. The set Γ contains m quality RCKs if there exist at least m RCKs, and otherwise it consists of all normal RCKs deduced from Σ. The algorithm is in $O(m(l+n)^3)$ time, where l is the length $|Y_1|$ ($|Y_2|$) of Y_1 (Y_2), and n is the size of Σ. In practice, m is often a *predefined* constant, and the algorithm is in *cubic-time*.

To determine whether Γ includes all normal RCKs that can be deduced from Σ, algorithm findRCKs leverages a notion of *completeness*, first studied for traditional

candidate keys in [LO78]. To present this notion we need the following notations. For pairs of lists (X_1, X_2) and (Z_1, Z_2),

- we denote by $(X_1, X_2) \setminus (Z_1, Z_2)$ the pair (X_1', X_2') obtained by removing elements of (Z_1, Z_2) from (X_1, X_2); that is, for any $(A, B) \in (Z_1, Z_2)$, $(A, B) \notin (X_1', X_2')$;
- we also define $(X_1, X_2) \cup (Z_1, Z_2)$ by adding elements of (Z_1, Z_2) to (X_1, X_2);
- similarly, we can define $(X_1, X_2, C) \setminus (Z_1, Z_2, C')$, for relative keys.

Consider an RCK $\gamma = (X_1, X_2, C)$ and an MD ϕ defined as $\bigwedge_{j\in[1,k]}(R_1[W_1[j]] \approx_j R_2[W_2[j]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$. We define $\mathsf{apply}(\gamma, \phi)$ to be the relative key (X_1', X_2', C'), where

$$(X_1', X_2') = ((X_1, X_2) \setminus (Z_1, Z_2)) \cup (W_1, W_2),$$

i.e., by removing from (X_1, X_2) pairs of $\mathsf{RHS}(\phi)$ and adding pairs of $\mathsf{LHS}(\phi)$; and C' is obtained from C by removing corresponding operators for attributes in $\mathsf{RHS}(\phi)$ and adding those for each pair in $\mathsf{LHS}(\phi)$. Intuitively, $\mathsf{apply}(\gamma, \phi)$ is a relative key deduced by "applying" MD ϕ to γ.

A nonempty set Γ of RCKs is said to be *complete w.r.t.* Σ if for each normal RCK γ in Γ and each MD ϕ in Σ, there exists a RCK γ_1 in Γ such that either $\gamma_1 \preceq \mathsf{apply}(\gamma, \phi)$ or $\gamma_1 = \mathsf{apply}(\gamma, \phi)$ (recall the notion $\preceq$ from Section 6.2.2).

That is, for all normal RCKs that can be deduced by possible applications of MDs in Σ, they are covered by "smaller" RCKs that are already in the set Γ.

This notion of completeness allows us to check whether Γ consists of all normal RCKs deduced from Σ. As will be seen shortly, our algorithm uses this property to determine whether or not Γ needs to be further expanded. To simplify the discussion, we also include in Γ the default relative key $(Y_1, Y_2, [=, \ldots, =])$, denoted by γ_0.

Proposition 6.5.1 *When* Γ *includes* γ_0, Γ *consists of all normal* RCKs *deduced from* Σ *if and only if* Γ *is complete* w.r.t. Σ.

Proof: First assume that Γ consists of all normal RCKs that can be deduced from Σ. Then for any normal RCK γ in Γ and each MD ϕ in Σ, $\gamma_1 = \mathsf{apply}(\gamma, \phi)$ is a normal relative key. Since Γ consists of all normal RCKs, for each RCK $\gamma_2 \preceq \gamma_1$, γ_2 is in Γ. Hence Γ is complete *w.r.t.* Σ.

Conversely, assume that Γ is complete *w.r.t.* Σ. We show that for any normal RCK γ such that $\Sigma \models_m \gamma$, γ can be deduced by repeated uses of the apply operator on normal RCKs in Γ and MDs in Σ. This suffices. For if it holds, then γ must be in Γ since Γ is complete.

Algorithm findRCKs

Input: Number m, a set Σ of MDs, and pairwise comparable (Y_1, Y_2).

Output: A set Γ of at most m RCKs.

1. $c := 0$; $S :=$ pairing (Σ, Y_1, Y_2);
2. **let** $\mathsf{ct}(R_1[A], R_2[B]) := 0$ for each $(R_1[A], R_2[B]) \in S$;
3. $\gamma_0 := (Y_1, Y_2, C)$, where $|C| = |Y_1|$ and C consists of equality '=' only;
4. $\Gamma := \{\gamma_0\}$;
5. **for** each RCK $\gamma \in \Gamma$ **do**
6. $L_\Sigma :=$ sortMD(Σ);
7. **for** each ϕ in L_Σ in the ascending order **do**
8. $L_\Sigma := L_\Sigma \setminus \{\phi\}$;
9. $\gamma' :=$ apply(γ, ϕ); flag := true;
10. **for** each $\gamma_1 \in \Gamma$ **do**
11. flag := flag and $(\gamma_1 \not\preceq \gamma')$;
12. **if** flag **then**
13. $\gamma' :=$ minimize (γ', Σ); $\Gamma := \Gamma \cup \{\gamma'\}$;
14. $c := c + 1$; incrementCt(S, γ'); $L_\Sigma :=$ sortMD(L_Σ);
15. **if** $c = m + 1$ **then return** $\Gamma \setminus \{\gamma_0\}$;
16. **return** Γ.

Procedure minimize $((X_1, X_2, C), \Sigma)$

Input: Relative key $\gamma = (X_1, X_2, C)$ and a set Σ of MDs.

Output: An RCK.

1. $L :=$ sort (X_1, X_2, C);
2. **for** each V=$(R_1[A], R_2[B], \approx)$ in L in the descending order **do**
3. **if** $\Sigma \models_m \gamma \setminus V$ /* using algorithm MDClosure */
4. **then** $\gamma := \gamma \setminus V$;
5. **return** γ;

Figure 6.7: Algorithm findRCKs

Since $\Sigma \models_m \gamma$, γ must be in the closure $(\Sigma, \gamma_0)^+$ (recall the notion of closures from the proof of Lemma 6.3.3). Since γ is normal, the deduction of γ uses only rules MD_1–MD_4 in the inference system $\mathcal{I}$. Since these rules correspond to the apply operation, γ can be deduced from normal RCKs in Γ and MDs in Σ by the apply operations. This can be verified by induction on the steps when γ is included in $(\Sigma, \gamma_0)^+$ by the chase process given in the proof of Lemma 6.3.3. □

Algorithm findRCKs. We are now ready to present Algorithm findRCKs, as shown in Fig. 6.7. Before we illustrate its details, we first present the procedures it uses.

(a) Procedure minimize takes as input Σ and a relative key $\gamma = (X_1, X_2, C)$ such that $\Sigma \models_m \gamma$, where γ is not necessarily an RCK; it returns an RCK by minimizing γ. It first sorts $(R_1[A], R_2[B], \approx)$ in γ based on $\mathsf{cost}(R_1[A], R_2[B])$ (line 1). It then processes each $(R_1[A], R_2[B], \approx)$ in the *descending* order, starting from the *most costly* one (line 2). More specifically, it *removes* $V = (R_1[A], R_2[B], \approx)$ from γ, as long as $\Sigma \models_m \gamma \setminus V$ (lines 3-4). Thus when the process terminates, it produces γ', an RCK such that $\Sigma \models_m \gamma'$. The deduction is checked by invoking algorithm MDClosure (Section 6.4).

(b) Procedure incrementCt (not shown) takes as input a set S of attribute pairs and an RCK γ. For each pair $(R_1[A], R_2[B])$ in S and γ, it increases $\mathsf{ct}(R_1[A], R_2[B])$ by 1.

(c) Procedure sortMD (not shown) sorts MDs in Σ based on the sum of the costs of their LHS attributes. The sorted MDs are stored in a list L_Σ, in ascending order.

We now present the main driver of Algorithm findRCKs. The algorithm uses a counter c to keep track of the number of RCKs in Γ, initially set to 0 (line 1). It first collects in S all pairs $(R_1[A], R_2[B])$ that are either in (Y_1, Y_2) or in some MD of Σ (referred to as $\mathsf{pairing}(\Sigma, Y_1, Y_2)$, line 1). The counters of these pairs are set to 0 (line 2). It then adds the default relative key $\gamma_0 = (Y_1, Y_2, C)$ to Γ (lines 3-4).

After these initialization steps, findRCKs repeatedly checks whether Γ is complete *w.r.t.* Σ. If not, it expands Γ (lines 5-15). More specifically, for each $\gamma \in \Gamma$ and $\phi \in \Sigma$, it inspects the condition for the completeness (lines 7-11). If Γ is not complete, an RCK γ' is added to Γ, where γ' is obtained by first applying ϕ to γ and then invoking minimize. The algorithm increases the counter c by 1, and re-sorts MDs in Σ based on the updated costs (lines 12-14).

The process proceeds until either Γ contains m RCKs (line 15; excluding the default key γ_0, which may not be a RCK), or it cannot be further expanded (line 16). In the latter case, Γ already *includes all* the normal RCKs that can be deduced from Σ, as verified by Proposition 6.5.1.

The algorithm deduces RCKs defined with attributes of low costs. Indeed, it sorts MDs in Σ based on their costs, and applies low-cost MDs first (lines 6-7). Moreover, it *dynamically adjusts* the costs after each RCK γ' is added, by increasing $\mathsf{ct}(R_1[A], R_2[B])$ of each $(R_1[A], R_2[B])$ in γ' (lines 4, 14). Further, Procedure minimize retains attributes pairs with low costs in RCKs and removes those of high costs.

Example 6.5.1: Consider MDs Σ_c described in Example 6.3.5, and attribute lists (Y_c, Y_b) of Example 6.1.1. We illustrate how algorithm findRCKs computes a set of

RCKs relative to (Y_c, Y_b) from Σ_c. We fix $m = 6$, weights $w_1 = 1$ and $w_2 = w_3 = 0$.

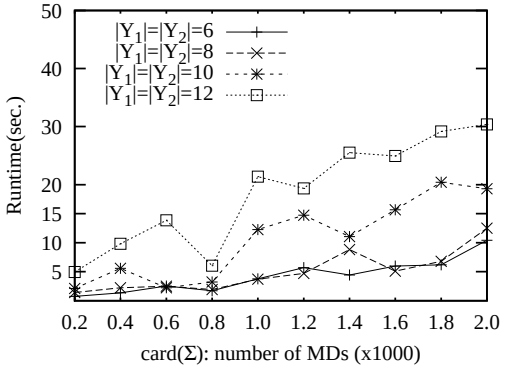
The table below shows how the following values are changed: (1) $\mathsf{cost}(R_1[A], R_2[B])$ for each pair $(R_1[A], R_2[B])$ appearing in Σ_c and (Y_c, Y_b), (2) the cost of each MD in Σ_c, and (3) the set Γ of RCKs deduced. When counter $c = 0$, the table only shows these values after step 4 of the algorithm. For $c \geq 1$, the values after step 15 are given.

attribute pairs/MDs	**counter c**				
	0	1	2	3	4
$\mathsf{cost}(\text{LN}, \text{LN})$	0	1	2	2	2
$\mathsf{cost}(\text{FN}, \text{FN})$	0	1	2	2	2
$\mathsf{cost}(\mathsf{addr}, \mathsf{post})$	0	1	1	2	2
$\mathsf{cost}(\mathsf{tel}, \mathsf{phn})$	0	0	1	1	2
$\mathsf{cost}(\mathsf{email}, \mathsf{email})$	0	0	0	1	2
$\mathsf{cost}(Y_c, Y_b)$	1	1	1	1	1
$\mathsf{cost}(\mathsf{LHS}(\varphi_1))$	0	3	5	6	6
$\mathsf{cost}(\mathsf{LHS}(\varphi_2))$	0	0	1	1	2
$\mathsf{cost}(\mathsf{LHS}(\varphi_3))$	0	0	0	1	2

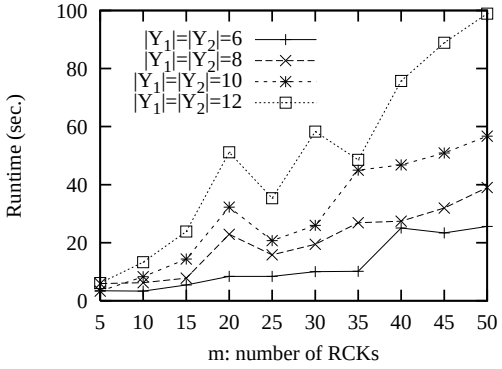
c	**new RCKs added to set Γ**
0	rck_0: $([Y_c], [Y_b], [=])$
1	rck_1: ([LN, addr, FN], [LN, post, FN], $[=, =, \approx_d]$)
2	rck_2: ([LN, tel, FN], [LN, phn, FN], $[=, =, \approx_d]$)
3	rck_3: ([email, addr], [email, post], $[=, =]$)
4	rck_4: ([email, tel], [email, phn], $[=, =]$)

The algorithm deduces RCKs as follows. (a) When $c = 0$, it applies MD φ_1 to rck_0 and gets rck_1. (b) When $c = 1$, rck_2 is deduced by applying φ_2 to rck_1. (c) When $c = 2$, rck_3 is deduced from φ_3 and rck_1. (d) When $c = 3$, rck_4 is found by applying φ_2 to rck_3. (e) When $c \geq 4$, nothing is changed since no new RCKs can be found. In fact the process terminates when $c = 4$ since no more RCKs are added to Γ, and all MDs in Σ have been checked against RCKs in Γ. The final set Γ is $\{\mathsf{rck}_1, \mathsf{rck}_2, \mathsf{rck}_3, \mathsf{rck}_4\}$. Note that rck_0 is not returned, since it is not an RCK. In the process the MD with the lowest cost is always chosen first. □

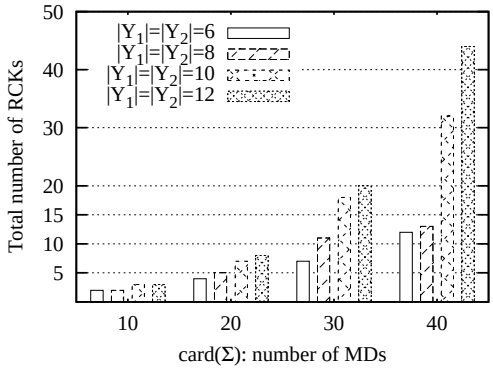
Complexity analysis. Let l be the length of (Y_1, Y_2) and n be the size of Σ. Observe the following. (a) The outer loop (line 5) of findRCKs executes at most m iterations. (b) In each iteration, $\mathsf{sortMD}(\Sigma)$ (line 6) takes $O(n \log n)$ time. (c) The innermost loop (lines 10–11) takes $O(n|\Gamma|)$ time *in total*. (d) Procedure minimize is invoked at most m times *in total*, which in turns calls MDClosure at most $O(|\gamma|)$ times (line 13),



(a) Scalability *w.r.t.* the number of MDs



(b) Scalability *w.r.t.* the number of RCKs



(c) The total number of RCKs

Figure 6.8: Scalability of Algorithm findRCKs

where $|\gamma| \leq l+n$. Thus the total cost of running MDClosure is in $O(m(n+l)^3)$ time (by Theorem 6.4.1, for fixed schemas). (e) $|\Gamma| \leq m(l+n)$. Putting these together, algorithm findRCKs is in $O(m(l+n)^3)$ time.

We remark that the algorithm is efficient in practice because it is run at compile time, m is often a small constant, and n and l are much smaller than the size of data relations.

6.6 Experimental Evaluation

In this section we present an experimental study of our techniques. We conducted four sets of experiments. The focus of the first set of experiments is on the scalability of algorithms findRCKs and MDClosure. Using data taken from the Web, we then evaluate the utility of RCKs in record matching. More specifically, in experiments 2 and 3 we evaluate the impacts of RCKs on the performance and accuracy of statistical and rule-based matching methods, respectively. Finally, the fourth set of experiments

demonstrates the effectiveness of RCKs in blocking and windowing.

We have implemented findRCKs, MDClosure, and two matching methods: sorted neighborhood [HS95] and Fellegi-Sunter model [FS69, Jar89] with expectation maximization (EM) algorithm for assessing parameters, in Java. The experiments were run on a machine with a Quad Core Xeon (2.8GHz) CPU and 8GB of memory. Each experiment was repeated over 5 times and the average is reported.

6.6.1 The Scalability of findRCKs and MDClosure

The first set of experiments evaluates the efficiency of algorithms findRCKs and MDClosure. Since the former makes use of the latter, we just report the results for findRCKs.

Given a set Σ of MDs, a number m, and pairwise compatible lists (Y_1, Y_2) over schemas (R_1, R_2), algorithm findRCKs finds a set of m candidate keys relative to (Y_1, Y_2) if there exist m RCKs. We investigated the impact of the cardinality $\mathsf{card}(\Sigma)$ of Σ, the number m of RCKs, and the length $|Y_1|$ (equivalently $|Y_2|$) of Y_1 on the performance of findRCKs.

The MDs used in these experiments were produced by a generator. Given schemas (R_1, R_2) and a number l, the generator randomly produces a set Σ of l MDs over the schemas.

Fixing $m = 20$, we varied $\mathsf{card}(\Sigma)$ from 200 to 2,000 in 200 increments, and studied its impact on findRCKs. The result is reported in Fig. 6.8(a), for $|Y_1|$ ranging over 6, 8, 10 and 12. We then fixed $\mathsf{card}(\Sigma) = 2,000$ and varied the number m of RCKs from 5 to 50 in 5 increments. We report in Fig. 6.8(b) the performance of findRCKs for various m and $|Y_1|$. Figures 6.8(a) and 6.8(b) tell us that findRCKs scales well with the number of MDs, the number of RCKs and the length $|Y_1|$. These results also show that the larger $|Y_1|$ is, the longer it takes, as expected.

We have also inspected the quality of RCKs found by findRCKs. We find that these RCKs are reasonably diverse when the weights w_1, w_2, w_3 used in our quality model (Section 6.5) are 1, and $\mathsf{ac}(R_1[A], R_2[B]) = 1$ for all attribute pairs. We also used these cost parameters in the other experiments.

Figure 6.8(c) reports the total number of RCKs derived from small sets Σ. It shows that when there are not many MDs available, we can still find a reasonable number of RCKs that, as will be seen below, suffice to direct quality matching.

6.6.2 Improvement on the Quality and Efficiency

The next three sets of experiments focus on the effectiveness of RCKs in record matching, blocking and windowing.

Experimental setting. We used an extension of the credit and billing schemas (Section 6.1), also referred to as credit and billing, which have 13 and 21 attributes, respectively. We picked a pair (Y_1, Y_2) of lists over (credit, billing) for identifying card holders. Each of the lists consists of 11 attributes for name, phone, street, city, county, zip, etc. The experiments used 7 simple MDs over credit and billing, which specify matching rules for card holders.

We populated instances of these schemas using real-life data, and introduced duplicates and noises to the instances. We evaluated the ability of our MD-based techniques to identify the duplicates. More specifically, we scraped addresses in the US from the Web, and sale items (books, DVDs) from online stores. Using the data we generated datasets controlled by the number K of credit and billing tuples, ranging from 10k to 80k. We then added 80% of duplicates, by copying existing tuples and changing some of their attributes that are not in Y_1 or Y_2. Then more errors were introduced to each attribute in the duplicates, including those in Y_1 and Y_2, with probability 80%, ranging from small typographical changes to complete change of the attribute.

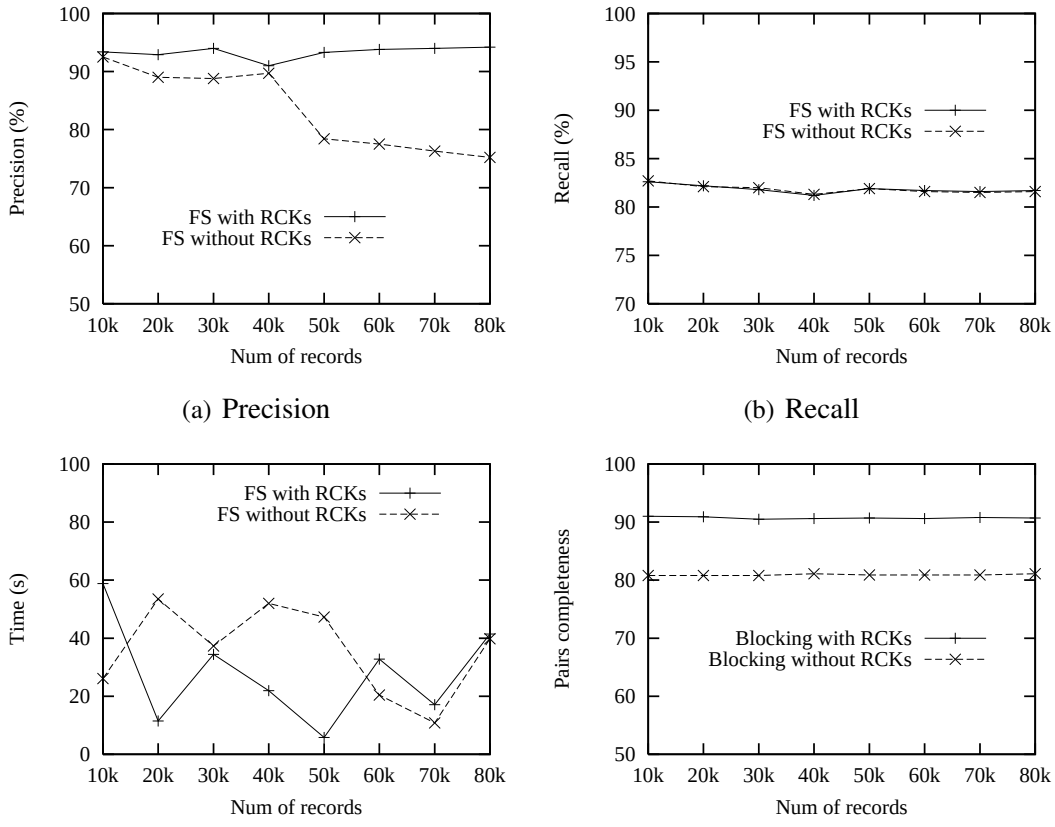
We used the DL metric (Damerau-Levenshtein) [GFS+01] for similarity test, defined as the minimum number of single-character insertions, deletions and substitutions required to transform a value v to another value v'. We used the implementation $\asymp_\theta$ of the DL-metric provided by SimMetrics (*http://www.dcs.shef.ac.uk/~sam/simmetrics.html*). For any values v and v', $v \asymp_\theta v'$ if the DL distance between v and v' is no more than $(1-\theta)\%$ of $\max(|v|, |v'|)$. In all the experiments we fixed $\theta = 0.8$.

To measure the quality of matches we used (a) *precision*, the ratio of *true matches* (true positive) correctly found by a matching algorithm to all the duplicates found, and (b) *recall*, the ratio of true matches correctly found to all the duplicates in the dataset.

To measure the benefits of blocking (windowing), we use s_M and s_U to denote the number of matched and non-matched pairs with blocking (windowing), and similarly, n_M and n_U for matched and non-matched pairs without blocking (windowing). We then define the *pairs completeness ratio* PC and the *reduction ratio* RR as follows:

$$\mathsf{PC} = s_M/n_M, \qquad \mathsf{RR} = 1 - (s_M + s_U)/(n_M + n_U).$$

Intuitively, the larger PC is, the more effective the blocking (windowing) strategy is. In addition, RR indicates the saving in comparison space.



(a) Precision (b) Recall

(c) Run time (d) Blocking

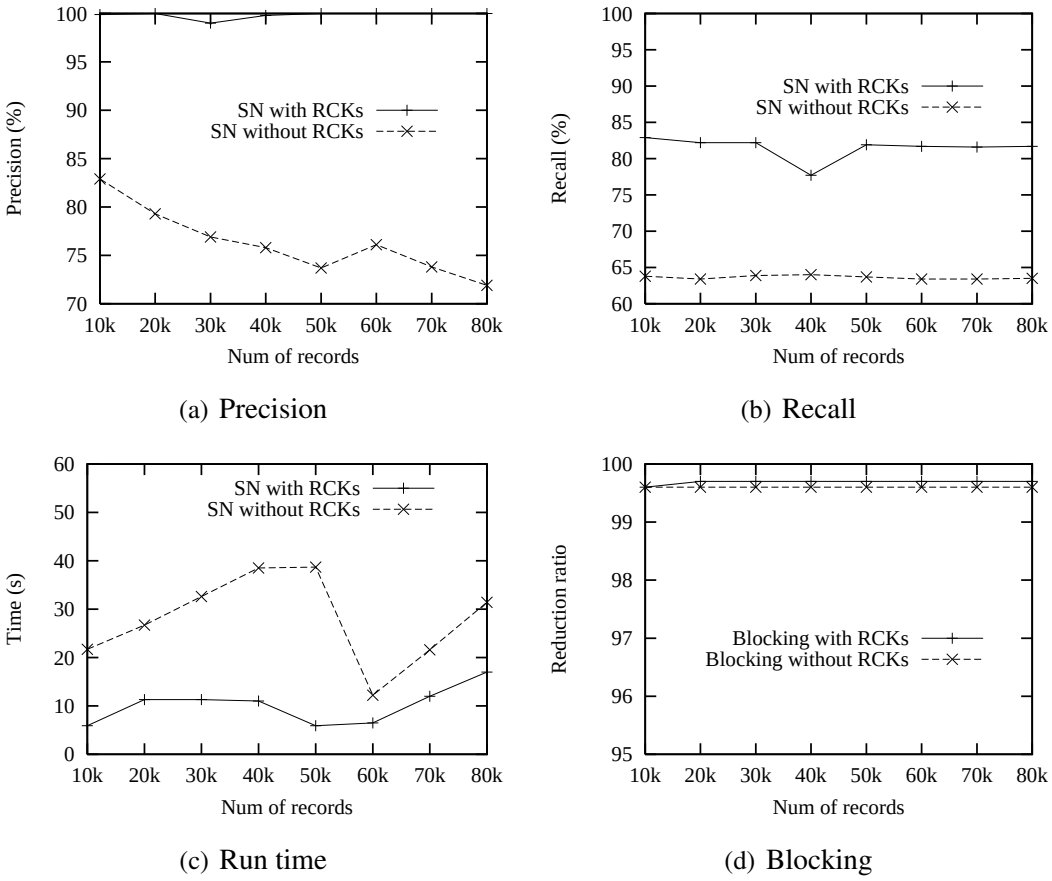
Figure 6.9: Fellegi-Sunter method

As the noises and duplicates in the datasets were introduced by the generator, precision, recall, PC and RR can be accurately computed from the results of matching, blocking and windowing by checking the truth held by the generator.

Experiments 2 and 3 employed windowing to improve efficiency, with a fixed window size of 10 (*i.e.,* the sliding window contained no more than 10 tuples). The same set of windowing keys was used in all these experiments to assure that the evaluation was fair.

Exp-2: Fellegi-Sunter method (FS) **[FS69]**. This statistical method is widely used to process, *e.g.,* census data. This set of experiments used FS to find matches, based on two comparison vectors: (a) one was the union of top five RCKs derived by our algorithms; and (b) the other was picked by an expectation maximization algorithm on a sample of at most 30k tuples. The EM algorithm is a powerful tool to automatically estimate parameters such as weights and threshold [Jar89]. We evaluated the performance of FS using these vectors, denoted by FS and $\mathsf{FS}_{\mathsf{rck}}$, respectively.

Accuracy. Figures 6.9(a) and (b) report the accuracy of FS and $\mathsf{FS}_{\mathsf{rck}}$, when the number



(a) Precision (b) Recall

(c) Run time (d) Blocking

Figure 6.10: Sorted Neighborhood method

K of tuples ranged from 10k to 80k. The results tell us that $\mathsf{FS}_{\mathsf{rck}}$ performs better than FS in precision, by 20% when K = 80k. Furthermore, $\mathsf{FS}_{\mathsf{rck}}$ is less sensitive to the size of the data: while the precision of FS decreases when K gets larger, $\mathsf{FS}_{\mathsf{rck}}$ does not. Observe that $\mathsf{FS}_{\mathsf{rck}}$ and FS have almost the same recalls. This shows that RCKs effectively improve the precision (increasing the number of true positive matches) without lowering the recall (without increasing the number of false positive matches).

In these experiments we also found that a single RCK tended to yield a lower recall, because any noise in the RCK attributes might lead to a miss-match. This is mediated by using the union of several RCKs, such that miss-matches by some RCKs could be rectified by the others. We found that $\mathsf{FS}_{\mathsf{rck}}$ became far less sensitive to noises when the union of RCKs was used.

Efficiency. As shown in Fig. 6.9(c), $\mathsf{FS}_{\mathsf{rck}}$ and FS have comparable performance. That is, RCKs do not incur extra cost while they may substantially improve the accuracy.

Exp-3: Sorted Neighborhood method (SN) **[HS95].** This is a popular rule-based method, which uses (a) rules of equational theory to guide how records should be

compared, and (b) a sliding window to improve the efficiency. However, the quality of rule-based methods highly depends on the skills of domain experts to get a good set of rules. We run SN on the same dataset as Exp-2, based on two sets of rules: (a) the 25 rules used in [HS95], denoted by SN; (b) the union of top five RCKs derived by our algorithms, denoted by $\mathsf{SN}_{\mathsf{rck}}$.

Accuracy. The results on match quality are reported in Figures 6.10(a) and 6.10(b), which show that $\mathsf{SN}_{\mathsf{rck}}$ consistently outperforms SN in both precision and recall, by around 20%. Observe that the precision of SN slightly decreases when K increases. In contrast, $\mathsf{SN}_{\mathsf{rck}}$ is less sensitive to the size of the data when precision and recall are concerned.

Efficiency. As shown in Fig. 6.10(c), $\mathsf{SN}_{\mathsf{rck}}$ consistently performs better than SN. This shows that RCKs effectively reduce comparisons (the number of attributes compared, and the number of rules applied), without decreasing the accuracy. Furthermore, the results tell us that both $\mathsf{SN}_{\mathsf{rck}}$ and SN scale well with the size of dataset.

Exp-4: Blocking and windowing. To evaluate the effectiveness of RCKs in blocking, we conducted experiments using the same dataset as before, and based on two blocking keys. One key consists of three attributes in top two RCKs derived by our algorithms. The other contains three attributes manually chosen. In both cases, one of the attributes is name, encoded by Soundex [Sou] before blocking.

The results for pairs completeness PC and reduction ratios RR are shown in Fig. 6.9(d) and Fig. 6.10(d), respectively (recall that the PC and RR can be computed by referencing the truth held by the data generator, without relying on any particular matching method). The results tell us that blocking keys based on partially encoded attributes in RCKs often yield comparable reduction ratios; at the same time, they lead to substantially better pairs completeness. Indeed, the improvement is consistently above 10%. We also conducted experiments to evaluate the effectiveness of RCKs in windowing, and found results comparable to those reported in Fig. 6.9(d) and Fig. 6.10(d).

Summary. From the experimental results we find the following. (a) Algorithms findRCKs and MDClosure scale well and are efficient. It takes no more than 100 seconds to deduce 50 quality RCKs from a set of 2000 MDs. (b) RCKs improve both the precision and recall of the matches found by FS and SN, and in most cases, improve the efficiency as well. For instance, it outperforms SN by around 20% in both precision and recall, and up to 30% in performance. Furthermore, using RCKs as comparison vectors, FS and SN become less sensitive to noises. (c) Using partially encoded

RCK attributes as blocking or windowing keys consistently improves the accuracy of matches found.

6.7 Related Work

A variety of methods (*e.g.,* [ACG02, CCGK07, CR02, FH76, FS69, GFS+01, GKMS04, Jar89, HS95, LSPR96, SB02, VEH02, Win02, Win04]) and tools (*e.g.,* Febrl, TAILOR, WHIRL, BigMatch) have been developed for record matching (see [EIV07] for a recent survey). There has also been a host of work on more general data cleaning and ETL tools (see [BS06] for a survey). This work is not to provide another record matching algorithm. Instead, it *complements* prior matching methods by providing dependency-based reasoning techniques to help decide keys for *matching, blocking* or *windowing*. An automated reasoning facility will effectively reduces manual effort and improves match quality and efficiency. While such a facility should logically become part of the record matching process, we are not aware of analogous functionality currently in any systems or tools.

Rules for matching are studied in [ACG02, ACK08, ARS09, CSGK07, HS95, LSPR96, SD05, SLD05, WNJ+08]. A class of rules is introduced in [HS95], which can be expressed as relative candidate keys of this work; in particular, the key used in Example 6.1.1 is borrowed from [HS95]. Extensions of [HS95] are proposed in [ACG02, ACK08], by supporting dimensional hierarchies and constant transformations to identify domain-specific abbreviations and conventions (*e.g.,* "United States" to "USA"). It is shown that matching rules and keys play an important role in industry-scale credit checking [WNJ+08]. The need for dependencies for record matching is also highlighted in [CSGK07, SLD05]. A class of *constant* keys is studied in [LSPR96], to match records in a single relation. Recursive algorithms are developed in [ARS09, SD05], to compute matches based on certain dependencies. The AJAX system [GFS+01] also advocates matching transformations specified in a declarative language. However, to the best of our knowledge, no previous work has formalized matching rules or matching keys as dependencies in a logic framework, or has studied automated techniques and inference systems for reasoning about dependencies for record matching. This work provides the first formal specifications and static analysis of matching rules, to deduce keys for matching, blocking and windowing via automated reasoning of dependencies. It should be mentioned that the idea of this work was presented in an invited tutorial [Fan08], without revealing technical details.

Another approach to deciding what attributes are important in comparison is based on probabilistic models, using an expectation maximization (EM) algorithm [Jar89, Win02]. In contrast, this work decides what attributes to compare by the static analysis of MDs at the schema level and at compile time. As will be seen in Section 6.6, the MD-based method outperforms the EM-based approach in both accuracy and efficiency. On the other hand, the two approaches *complement* each other: one can first *discover* a small set of MDs via sampling and learning, and then leverage the reasoning techniques to deduce RCKs. It should be remarked to get an initial set of MDs one can also leverage *domain knowledge analysis*, along the same lines as the design of FDs.

Dependency theory is almost as old as the study of relational databases itself. Traditional dependencies, *e.g.,* FDs, are first-order logic sentences in which domain-specific similarity metrics are *not* expressible. Furthermore, these dependencies are static constraints for which updates are not a concern, and are studied for schema design on clean data (see, *e.g.,* [AHV95] for a detailed discussion of relational dependencies). In contrast, for matching records from unreliable data sources one needs similarity metrics to accommodate errors in the data. In addition, as will be seen shortly, the static semantics of traditional dependencies is no longer appropriate in record matching. Indeed, the semantics of MDs and the notion of their deductions are a departure from their traditional counterparts for dependencies and implication.

There have been extensions of FDs by supporting similarity metrics [BV06, KSSV09]. There has also be been work on schema design for uncertain relations by extending FDs [ADS09]. Like traditional FDs, these extensions are defined on a single relation and have a static semantics. They are quite different from MDs studied in this work, which are defined across possibly different relations and have a dynamic semantics.

Dynamic constraints have been studied for database evolution [Via87] and for XML updates [CAM07]. These constraints aim to express an invariant connection between the old value and the new value of a data element when the data is updated. They differ from MDs in that they are restrictions on how given updates should be carried out. In contrast, MDs specify how data elements should be identified for record matching. In other words, MDs are to determine what (implicit) updates are necessary for identifying records. Furthermore, similarity metrics are not supported by the constraints of [CAM07, Via87].

Chapter 7

Towards Certain Fixes with Editing Rules and Master Data

A variety of integrity constraints have been studied for data cleaning. While these constraints can detect the presence of errors, they fall short of guiding us to correct the errors. Indeed, data repairing based on these constraints may not find *certain fixes* that are absolutely correct, and worse, may introduce new errors when repairing the data. In this chapter 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 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. We also provide an algorithm to identify minimal certain regions, such that a certain fix is warranted by editing rules and master data as long as one of the regions is correct. We experimentally verify the effectiveness and scalability of the algorithm.

7.1 Introduction

Real-life data is often dirty: 1%–5% of business data contains errors [Red98]. Dirty data costs US companies alone 600 billion dollars each year [Eck02]. These highlight the need for data cleaning, to catch and fix errors in the data. Indeed, the market for data cleaning tools is growing at 17% annually, way above the 7% average forecast for other IT sectors [Gar07].

	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 and t_2

	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.	Lnd	NW1 6XE	25/12/67	M

(b) Example master relation D_m

Figure 7.1: Example input tuples and master relation

An important functionality that is expected from a data cleaning tool is *data monitoring* [CGGM03, SMO07]: it is to find and correct errors in a tuple when it is created, either entered manually or generated by some process. That is, we want to ensure that a tuple t is clean before it is used, to prevent errors introduced by adding t. As noted by [SMO07], it is far less costly to correct a tuple at the point of entry than fixing it afterward.

A variety of integrity constraints have been studied 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]). These constraints help us determine whether data is dirty or not, *i.e.,* whether errors are present in the data. However, they fall short of telling us which attributes of t are erroneous and moreover, how to correct the errors.

Example 7.1.1: Consider an input tuple t_1 given in Fig. 7.1(a). It specifies a supplier in the UK: name (FN, LN), phone number (area code AC and phone phn), 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 if AC is 020, city should be Ldn, and when AC is 131, city must be Edi. These can be expressed as conditional functional dependencies (CFDs [FGJK08]). The tuple t_1 is *inconsistent*: $t_1[\mathsf{AC}] = 020$ but $t_1[\mathsf{city}] = \mathsf{Edi}$.

The CFDs detect that either $t_1[\mathsf{AC}]$ or $t_1[\mathsf{city}]$ is incorrect. However, they do not tell us which of the two attributes is wrong and to what value it should be changed. □

Several heuristic methods have been studied for repairing data based on constraints [ABC03b, BFFR05, CFG$^+$07, FH76, KL09, HSW09]. For the reasons mentioned above, however, these methods do not guarantee to find correct fixes in data monitoring; worse still, they may introduce new errors when trying to repair the data. For instance, tuple s_1 of Fig. 7.1(b) indicates corrections to t_1. Nevertheless, all of 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 motivates the quest for effective methods to find *certain fixes* that are guaranteed correct [Gil88, HSW09]. The need for this is evident in monitoring *critical* data, where an error may have disastrous consequences [HSW09]. To this end we need *editing rules* that tell us how to fix errors, *i.e.,* which attributes are wrong and what values they should take. In contrast, integrity 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 is being developed by IBM, SAP, Microsoft and Oracle. In particular, master data has been explored to provide *a data entry solution* in the SOA (Service Oriented Architecture) at IBM [SMO07], for data monitoring.

Example 7.1.2: A master relation D_m is shown in Fig. 7.1(b). Each tuple in D_m specifies a person in the UK in terms of the name, home phone (Hphn), mobile phone (Mphn), address, date of birth (DOB) and gender. An example editing rule is:

- eR_1: 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 user.

This rule makes *corrections* to attributes $t[\mathsf{AC}]$ and $t[\mathsf{str}]$, by taking values from master data s_1. Another editing rule is

- eR_2: 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 input tuple t_2 given in Fig. 7.1(a), in which attributes $t_2[\mathsf{str}, \mathsf{zip}]$ 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* $t[\mathsf{str}, \mathsf{zip}]$ by taking the corresponding values from the master tuple s_1. □

Contributions. We propose a method for data monitoring, by capitalizing on editing rules and master data.

(1) We introduce a class of editing rules defined in terms of data patterns and updates (Section 7.2). Given an input tuple t that matches a pattern, editing rules tell us what attributes of t should be updated and what values from master data should be assigned to them. In contrast to constraints, editing rules have a *dynamic* semantics, and are *relative to* master data. All the rules in Example 7.1.2 can be written as editing rules, but they are not expressible as constraints.

(2) We identify and study fundamental problems for reasoning about editing rules (Section 7.3). The analyses are relative to a *region* (Z, T_c), where Z is a set of attributes and T_c is a pattern tableau. One problem is to decide whether a set Σ of editing rules guarantees to find a *unique* (deterministic [Gil88, HSW09]) fix for input tuples t that match a pattern in T_c. The other problems concern whether Σ is able to fix all the attributes of such tuples. Intuitively, as long as $t[Z]$ is assured correct, we want to ensure that editing rules can find a certain fix for t. We show that these problems are coNP-complete, NP-complete or #P-complete, but we identify special cases that are in PTIME.

(3) We develop an algorithm to derive certain regions from a set Σ of rules and master data D_m (Section 7.4). A certain region (Z, T_c) is such a region that a certain fix is warranted for an input tuple t as long as $t[Z]$ is assured correct and t matches a pattern in T_c. We naturally want to recommend minimal such Z's to the users. However, we show that the problem for finding minimal certain regions is NP-complete. Nevertheless, we develop an efficient heuristic algorithm to find a set of certain regions, based on a quality model.

(4) We experimentally verify the effectiveness and scalability of the algorithm, using real-life hospital data, DBLP as well as synthetic data TPC-H and RAND (Section 7.5). We find that the algorithm scales well with the size of master data and the size of editing rules. We also show that certain regions automatically derived by the heuristic algorithm are comparable to certain regions manually designed, when they are used to clean input tuples.

Taken together, these yield a data entry solution. A set of certain regions are first recommended to the users, derived from editing rules and master data available. Then for any input tuple t, if the users ensure that any of those regions in t is correct, the rules guarantee to find a certain fix for t.

Related work. Several classes of constraints have been studied for data cleaning (*e.g.,* [ABC03b, BFFR05, CM05, BFM07, FGJK08, KL09, Wij05]; see [Fan08] for a survey). As remarked earlier, 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. They are also quite different from edits studied for census data repairing [FH76, Gil88, HSW09], which are conditions defined on a single record and are used to detect errors.

Closer to editing rules are matching dependencies (MDs [FJLM09]). We shall elaborate their differences in Section 7.2.

Rules have been studied for active databases (see [WC96] for a survey), and 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.

Data monitoring is advocated in [CGGM03, FPS$^+$10, SMO07]. A method for matching input tuples with master data was presented in [CGGM03], but it did not consider repairing the tuples. There has been a host of work on data repairing [ABC03b, BFFR05, CM05, BFM07, FGJK08, FH76, Gil88, HSW09, KL09, Wij05], aiming to find a consistent database D' that minimally differs from original data D. It is to repair a database rather than cleaning an input tuple at the point of entry. Although the need for finding certain fixes has long been recognized [Gil88, HSW09], prior methods do not guarantee that all the errors in D are fixed, or that D' does not have new errors. Master data is not considered in those methods.

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

7.2 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 use editing rules and data values in D_m.

We specify input tuples t with a relation schema R. We 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, which is often distinct from R. As remarked earlier, D_m can be assumed consistent and complete [RW08].

Editing rules. An *editing rule* (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 lists of distinct attributes in schemas R and R_m, respectively, where $|X| = |X_m|$,
- B is an attribute such that $B \in R \setminus X$, and $B_m \in R_m$,
- t_p is a pattern tuple over a set of distinct attributes X_p in R, where for each $A \in X_p$, $t_p[A]$ is either a or $\bar{a}$ for a constant a drawn from the domain of A.

We say that a tuple t of R *matches* pattern $t_p[X_p]$, denoted by $t[X_p] \approx t_p[X_p]$, if for each attribute $A \in X_p$, (a) $t[A] = a$ if $t_p[A]$ is a, and (b) $t[A] \neq a$ if $t_p[A]$ is $\bar{a}$.

Example 7.2.1: Consider the supplier schema R and master relation schema R_m shown in Fig. 6.1. The rules eR_1, eR_2 and eR_3 described in Example 7.1.2 can be expressed as the following editing rules 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 AC, str and 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 constraint. 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 $\mathsf{type} = 1$ (home phone) yet $\mathsf{AC} \neq 0800$ (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 master data. □

We next give the semantics of editing rules.

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

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, we can safely *enrich* $t[B]$ with master data $t_m[B_m]$ as long as (a) $t[X]$ and $t_m[X_m]$ are identified, and (b) $t[X_p]$ matches the pattern in φ. This yields a new tuple t' with $t'[B] = t_m[B_m]$ 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 7.2.2: As shown in Example 7.1.2, we can correct t_1 by applying the eRs φ_1 and master tuple s_1 to t_1. As a result, $t_1[\mathsf{AC}, \mathsf{str}]$ is changed to (131, 51 Elm Row).

Furthermore, we can normalize $t_1[\text{FN}]$ by applying φ_2 and s_1 to t_1, such that $t_1[\text{FN}]$ is changed from Bob to Robert.

The eRs φ_3 and master tuple s_1 can be applied to t_2, such that $t_2[\mathsf{city}]$ is corrected and $t_2[\mathsf{str}, \mathsf{zip}]$ is enriched. □

Remarks. (1) As remarked earlier, editing rules are quite different from CFDs [FGJK08]. A CFD $\psi = (X \to Y, t_p)$ is defined on a single relation R, where $X \to Y$ 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 $X \to Y$ is enforced on t_1 and t_2. It has a *static* semantics: t_1 and t_2 either satisfy or violate ψ, but they are not changed. 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.

(2) The MDs of [FJLM09] 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 a tuple t_1 of R_1 and a tuple t_2 of R_2, ϕ assures 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. In contrast to editing rules, (a) MDs are for record matching, not for data cleaning. They specify what attributes should be identified, but do not tell us how to update them. (b) MDs do not carry data patterns. (c) MDs do not consider master data, and hence, their analysis is far less challenging. Indeed, the static analyses of editing rules are intractable, while the analysis of MDs is in PTIME [FJLM09].

CFDs and MDs *cannot* be expressed as eRs, and *vice versa*.

(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.

7.3 Ensuring Unique and Certain Fixes

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, *i.e.,* (a) 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 (b) all the attributes of t' are ensured correct.

Below we first formalize the notion of certain fixes. We then study several problems for deciding whether Σ and D_m suffice to find a certain fix, *i.e.,* they ensure (a) and (b).

7.3.1 Certain Fixes and Certain Regions

When applying an eR φ and a master tuple t_m to t, we update t with a value in t_m. To ensure that the change makes sense, some attributes of t have to be assured correct. In addition, we cannot update t if either it does not match the pattern of φ or it cannot find a master tuple t_m in D_m that carries the information needed for correcting t.

Example 7.3.1: Consider the master relation D_m given in Fig. 7.1(a) and a set Σ_0 consisting of $\varphi_1, \varphi_2, \varphi_3$ and φ_4 of Example 7.2.1. Given input tuple t_3 of Fig. 7.1(a), both (φ_1, s_1) and (φ_3, s_2) apply to t_3. However, they suggest to update $t_3[\mathsf{city}]$ with distinct values Edi and Lnd. The conflict arises because $t_3[\mathsf{AC}]$ and $t_3[\mathsf{zip}]$ are inconsistent. Hence to fix t_3, we need to assure that one of $t_3[\mathsf{AC}]$ and $t_3[\mathsf{zip}]$ is correct.

Now consider tuple t_4 of Fig. 7.1(a). We find that no eRs in Σ_0 and tuples in D_m can be applied to t_4, and hence, we cannot decide 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 attributes in R, and T_c is a *pattern tableau* consisting of a set of pattern tuples with attributes in Z, such that for each tuple $t_c \in T_c$ and each attribute $A \in Z$, $t_c[A]$ is one of _ , a or $\bar{a}$. Here a is a constant in the domain of A, and _ is an unnamed variable (wildcard).

Intuitively, a region tells us that to correctly fix errors in a tuple t, $t[Z]$ should be assured correct, and $t[Z]$ should "satisfy" a pattern in T_c (defined below). Here T_c specifies what cases of input tuples are covered by eRs and D_m.

A tuple t of R *satisfies* a pattern tuple t_c in T_c, denoted by $t \rightleftharpoons t_c$, if for each $A \in Z$, either $t_c[A] = \_$, or $t[A] \approx t_c[A]$. That is, $t \rightleftharpoons t_c$ if either $t_c[A]$ is a wildcard, or $t[A]$ matches $t_c[A]$ when $t_c[A]$ is a or $\bar{a}$. We refer to t as a tuple *covered* by (Z, T_c) if there exists $t_c \in T_c$ such that $t \rightleftharpoons t_c$.

Consider an eR $\varphi = ((X, X_m) \to (B, B_m), t_p[X_p])$ and a master tuple t_m. We say that φ and t_m *correctly apply* to a tuple t *w.r.t.* (Z, T_c), denoted by $t \to_{((Z,T_c),\varphi,t_m)} t'$, if (a) $t \to_{(\varphi,t_m)} t'$, (b) $X \subseteq Z$, $X_p \subseteq Z$, $B \notin Z$, and (c) there exists a pattern tuple $t_c \in T_c$ such that $t \rightleftharpoons t_c$.

That is, it is justified to apply φ and t_m to t for those t covered by (Z, T_c) if $t[X, X_p]$ is correct. As $t[Z]$ is correct, we do not allow it to be changed by enforcing $B \notin Z$.

Example 7.3.2: Referring to Example 7.3.1, a region for tuples of R is $(Z_{\mathsf{AH}}, T_{\mathsf{AH}}) = ((\mathsf{AC}, \mathsf{phn}, \mathsf{type}), \{(\overline{0800}, \_, 2)\})$. Hence, if $t_3[\mathsf{AC}, \mathsf{phn}, \mathsf{type}]$ is correct, then (φ_3, s_2) can be correctly applied to t_3, yielding $t_3 \to_{((\mathsf{AC},\mathsf{phn}),T_{\mathsf{AC}},\varphi_3,s_2)} t'_3$, 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. □

Observe that if $t \rightarrow_{((Z,T_c),\varphi,t_m)} t'$, then $t'[B]$ is also assured correct. 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 $\mathsf{ext}(Z, T_c, \varphi)$.

For instance, $\mathsf{ext}((\mathsf{AC}, \mathsf{phn}, \mathsf{type}), T_{\mathsf{AH}}, \varphi_3)$ is (Z', T'), where Z' consists of $\mathsf{AC}, \mathsf{phn}, \mathsf{type}$, str, city and zip, and T' has a single tuple $t'_c = (\overline{0800}, \_, 2, \_, \_, \_)$.

Certain fix. For a tuple t of R covered by (Z, T_c), we want to make sure that we can get a unique fix t' no matter how eRs in Σ and tuples in D_m are applied to t.

We say that a tuple t' is a *fix* of t by (Σ, D_m), denoted by $t \rightarrow^*_{((Z,T_c),\Sigma,D_m)} t'$, if there exists a finite sequence $t_0 = t, t_1, \ldots, t_k = t'$ of tuples of R, and for each $i \in [1, k]$, there exist $\varphi_i \in \Sigma$ and $t_{m_i} \in D_m$ such that

(a) $t_{i-1} \rightarrow_{((Z_{i-1},T_{i-1}),\varphi_i,t_{m_i})} t_i$, where $(Z_0, T_0) = (Z, T_c)$, and $(Z_i, T_i) = \mathsf{ext}(Z_{i-1}, T_{i-1}, \varphi_i)$;
(b) $t_i[Z] = t[Z]$; and
(c) for all $\varphi \in \Sigma$ and $t_m \in D_m$, $t' \rightarrow_{((Z_m,T_m),\varphi,t_m)} t'$.

Intuitively, (a) each step of the correcting process is justified; (b) $t[Z]$ is assumed correct and hence, remains unchanged; and (c) t' is a fixpoint and cannot be further updated.

We say that 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, we refer to Z_m as the set of attributes of t *covered by* (Z, T_c, Σ, D_m).

The fix t' is called the *certain fix* if the set of attributes covered by (Z, T_c, Σ, D_m) includes *all* the attributes in R. Intuitively, if t has a certain fix t' then (a) it has a unique fix and (b) all the attributes of t' are correct provided that $t[Z]$ is correct. A notion of deterministic fix was addressed in [Gil88, HSW09]. It refers to unique fixes, *i.e.,* (a) above, without requiring (b). Further, it is not defined relative to (Z, T_c).

Example 7.3.3: By the set Σ_0 of eRs of Example 7.3.1 and the master data D_m of Fig. 7.1(b), tuple t_3 of Fig. 7.1(a) has a unique fix *w.r.t.* $(Z_{\mathsf{AH}}, T_{\mathsf{AH}})$, namely, t'_3 given in Example 7.3.2. However, as observed in Example 7.3.1, 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 7.2.2, tuple t_1 of Fig. 7.1(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. 7.1(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. □

Certain region. We say that (Z, T_c) is a *certain region* for (Σ, D_m) if for all tuples t of R that are covered 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 in Z) or by master data ($R \setminus Z$).

Example 7.3.4: As shown in Example 7.3.3, $(Z_{\mathsf{zm}}, T_{\mathsf{zm}})$ 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} with 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 tuples covered by the region, a certain fix is warranted. □

7.3.2 Reasoning about Editing Rules

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 covered by a region (Z, T_c). This motivates us to study several problems for reasoning about editing rules, and establish their complexity bounds.

The consistency problem. One problem is to decide whether (Σ, D_m) and (Z, T_c) do not have conflicts. We say that (Σ, D_m) is *consistent relative to* (Z, T_c) if for each input tuple t of R that is covered by (Z, T_c), t has a unique fix by (Σ, D_m) *w.r.t.* (Z, T_c).

Example 7.3.5: There exist (Σ, D_m) and (Z, T_c) that are inconsistent. Indeed, (Σ_0, D_m) is not consistent relative to $(Z_{\mathsf{AHZ}}, T_{\mathsf{AHZ}})$ of Example 7.3.3, since tuple 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* is to determine, given (Z, T_c) and (Σ, D_m), whether (Σ, D_m) is consistent relative to (Z, T_c).

Theorem 7.3.1: *The consistency problem is co*NP*-complete, even for relations with infinite-domain attributes only.* □

Proof: (1) We first show that the problem is in coNP, by providing an NP algorithm for its complement, *i.e.,* the algorithm returns 'yes' if and only if (Σ, D_m) is *not* consistent relative to (Z, T_c).

We define dom to be the set of all constants appearing in D_m and Σ, and introduce a variable v representing a distinct constant not in dom. It suffices to consider R tuples t such that for each attribute A of R, $t[A]$ is either a constant in dom or the variable v.

The NP algorithm works as follows:

(a) guess a tuple t_c in T_c;

(b) guess an R tuple t such that for each attribute A of R, $t[A]$ is a constant in dom or the variable v; and

(c) If $t \rightleftharpoons t_c$, then check whether (Σ, D_m) is consistent relative to $(Z, \{t[Z]\})$. If not, the algorithm returns 'yes'.

By Theorem 7.3.5 (see its proof below), checking whether (Σ, D_m) is consistent relative to $(Z, \{t[Z]\})$ is in PTIME since $t[Z]$ consists of only constants. Thus, the algorithm is in NP. It is routine to verify its correctness.

(2) We next show that the problem is coNP-hard, by reduction from the 3SAT problem to its complement.

An instance ϕ of 3SAT is of the form $C_1 \wedge \cdots \wedge C_n$ where all the variables in ϕ are $x_1, \ldots, x_m$, each clause C_j ($j \in [1,n]$) is of the form $y_{j_1} \vee y_{j_2} \vee y_{j_3}$, and moreover, for $i \in [1,3]$, y_{j_i} is either $x_{p_{ji}}$ or $\overline{x_{p_{ji}}}$ for $p_{ji} \in [1,m]$. Here we use $x_{p_{ji}}$ to denote the occurrence of a variable in the literal i of clause C_j. The 3SAT problem is to determine whether ϕ is satisfiable. The 3SAT problem is NP-complete (cf. [GJ79]).

Given an instance ϕ of the 3SAT problem, we construct the following: (a) schemas R and R_m, (b) a master relation D_m of schema R_m, (c) a pattern tableau T_c consists of a single tuple t_c for a set Z of attributes of schema R, and (d) a set Σ of eRs, such that (Σ, D_m) is consistent relative to (Z, T_c) if and only if the instance ϕ is *not* satisfiable.

(a) We define the schema $R(A, X_1, \ldots, X_m, C_1, \ldots, C_n, V, B)$ and the master schema $R_m(Y_{0_m}, Y_{1_m}, A_m, V_m, B_m)$ such that all attribute have an (infinite) integer domain.

(b) The master relation D_m contains three tuples:

	Y_{0_m}	Y_{1_m}	A_m	V_m	B_m
t_{m_1}:	0	1	1	1	1
t_{m_2}:	0	1	1	1	0
t_{m_3}:	0	1	1	0	1

(c) We define the $Z = (AX_1 \ldots X_m)$ and $t_c[Z] = (1, \_, \ldots, \_)$.

(d) The set Σ of eRs is $\Sigma_1 \cup \ldots \cup \Sigma_n \cup \Sigma_{C,V} \cup \Sigma_{V,B}$, where

- for each clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ ($j \in [1,n]$), we define a set Σ_j of eight eRs $\varphi_{j,(b_1 b_2 b_3)}$ in the form of $((A, A_m) \rightarrow (C_j, Y_{j_m}), t_{p,(b_1 b_2 b_3)}\ [X_{p_{j1}} X_{p_{j2}}\ X_{p_{j3}}] = (b_1, b_2, b_3))$, where $Y_{j_m} = Y_{0_m}$ if (b_1, b_2, b_3) makes C_j false, or $Y_{j_m} = Y_{1_m}$ if (b_1, b_2, b_3) makes C_j true. Here for each $i \in [1,3]$, $b_i \in \{0,1\}$, and (b_1, b_2, b_3) is the truth assignment ξ such that for $i \in [1,3]$, $\xi(x_{p_{ji}}) = b_i$ for ϕ.

- $\Sigma_{C,V} = \{\varphi_0, \ldots, \varphi_n\}$ consists of $n+1$ eRs, where $\varphi_0 = ((A, A_m) \to (V, V_m), t_{p_0}[C_1 \ldots C_n] = (1, \ldots, 1))$, and for $j \in [1, n]$, $\varphi_j = ((A, A_m) \to (V, V_m), t_{p_j}[C_j] = (0))$.
- $\Sigma_{V,B}$ consists of a single eR $\varphi_{V,B} = ((V, V_m) \to (B, B_m), ())$, *i.e.,* with an empty pattern tuple.

We now verify that (Σ, D_m) is consistent relative to (Z, T_c) if and only if the 3SAT instance ϕ is *not* satisfiable. First assume that (Σ, D_m) is consistent relative to (Z, T_c). We prove that ϕ is not satisfiable by contradictions. If ϕ is satisfiable, then there exists a truth assignment ξ of the variables $x_1, \ldots, x_m$ that makes ϕ true. Let t be a tuple of relation R such that $t[AX_1 \ldots X_m] = (1, \xi(x_1), \ldots, \xi(X_m))$, and $t[C_1 \ldots C_n V, B]$ can be any (partial) tuple. Then by applying the eRs in $\Sigma_1 \cup \ldots \cup \Sigma_n$ and the master tuple t_{m_1} (t_{m_2} or t_{m_3}) to the tuple t, we have another tuple t_1, where $t_1[AX_1 \ldots X_m] = t[AX_1 \ldots X_m]$ and $t_1[C_1 \ldots C_n] = (1, \ldots, 1)$. By applying φ_0 in $\Sigma_{C,V}$ and t_{m_1} or t_{m_2} to t_1, we have t_2, where $t_2[AX_1 \ldots X_m\ C_1 \ldots C_n] = t_1[AX_1 \ldots X_m\ C_1 \ldots C_n]$ and $t_2[V] = 1$. Moreover, (a) by applying the single eR in $\Sigma_{V,B}$ and t_{m_1} to t_2, we have $t_{3,1}$ such that $t_{3,1}[AX_1 \ldots X_m\ C_1 \ldots C_n V] = t_2[AX_1 \ldots X_m\ C_1 \ldots C_n V]$ and $t_{3,1}[B] = 1$; in contrast, (b) by applying the eR in $\Sigma_{V,B}$ and t_{m_2} to t_2, we have $t_{3,2}$, where $t_{3,2}[AX_1 \ldots X_m\ C_1 \ldots C_n V] = t_2[AX_1 \ldots X_m\ C_1 \ldots C_n V]$ and $t_{3,2}[B] = 0$. That is, (Σ, D_m) is not consistent relative to (Z, T_c), which contradicts our assumption.

Conversely, assume that the 3SAT instance ϕ is *not* satisfiable. We show that (Σ, D_m) is consistent relative to (Z, T_c).

Let t be an R tuple such that $t[A, X_1, \ldots, X_m]$ is initially correct. It suffices consider the following cases.

Case a. There exists $i \in [1, m]$ such that $t[X_i] \notin \{0, 1\}$. Thus, there must exist Σ_j ($1 \leq j \leq n$) such that all eRs in Σ_j and all master tuples in D_m cannot be applied t. In particular, the eR φ_0 in $\Sigma_{C,V}$ cannot be applied to the tuple t since the correct region cannot be expanded to include all attributes $C_1, \ldots, C_n$. For this case, it is easy to see that (Σ, D_m) is consistent relative to (Z, T_c).

Case b. For each $i \in [1, m]$, $t[X_i] \in \{0, 1\}$. Since ϕ is *not* satisfiable, the eR φ_0 in $\Sigma_{C,V}$ and any tuple in D_m cannot be applied to t. Therefore, (Σ, D_m) is consistent relative to (Z, T_c).

Hence (Σ, D_m) is consistent in all cases of R tuples. □

The consistency analysis of eRs is more intriguing than its CFD counterpart, which is NP-complete but is in PTIME when all attributes involved have an infinite domain [FGJK08]. It is also much harder than MDs, which is in quadratic-time [FJLM09].

Nevertheless, it is decidable, as opposed to the undecidability for reasoning about rules for active databases [WC96].

The coverage problem. Another problem is to determine whether (Σ, D_m) is able to fix errors in all attributes of input tuples that are covered by (Z, T_c).

The *coverage problem* is to determine, given any (Z, T_c) and (Σ, D_m), whether (Z, T_c) is a certain region for (Σ, D_m).

No matter how desirable to find certain regions, the coverage problem is intractable, although it is decidable.

Theorem 7.3.2: *The coverage problem is co*NP*-complete.* □

Proof: (1) We first show that the problem is in coNP by giving a coNP algorithm. The algorithm is the same as the one developed in the proof of Theorem 7.3.1, except that in the last step, it uses a variation of the PTIME algorithm given in Theorem 7.3.5 such that it only returns 'yes' if both the set S is empty, and the tuple t is wildcard free.

(2) We then show that the problem is coNP-hard by reduction from the 3SAT problem to its complement. Given an instance ϕ of the 3SAT problem, we construct schemas R and R_m, a master relation D_m of R_m, a set $Z \cup \{B\}$ of attributes of R, and a set Σ of eRs such that (Z, T_c) is a certain region for (Σ, D_m) if and only if ϕ is *not* satisfiable.

(a) The schema R_m, its master relation D_m, R, Z, and the pattern tableau T_c are the same as their corresponding ones defined in the proof of Theorem 7.3.1.

(b) We define the $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n \cup \Sigma_{C,V} \cup \Sigma_{V,B}$, where

- for each clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ ($j \in [1, n]$), we define a set Σ_j of eight eRs in the form of $\varphi_{j,(b_1 b_2 b_3)} = ((A, A_m) \rightarrow (C_j, Y_{j_m}), t_{p,(b_1 b_2 b_3)}[X_{p_{j1}} X_{p_{j2}} X_{p_{j3}}])$, where $t_{p,(b_1 b_2 b_3)}[X_{p_{ji}}] = \bar{1}$ if $b_i = 0$ and $t_{p,(b_1 b_2 b_3)}[X_{p_{ji}}] = 1$ if $b_i = 1$, $Y_{j_m} = Y_{0_m}$ if (b_1, b_2, b_3) makes C_j false, and $Y_{j_m} = Y_{1_m}$ if (b_1, b_2, b_3) makes C_j true.
  Here (b_1, b_2, b_3) is the truth assignment ξ such that for each $i \in [1, 3]$, $b_i \in \{0, 1\}$ and $\xi(X_{p_{ji}}) = b_i$.
- $\Sigma_{C,V}$ and $\Sigma_{V,B}$ are the same as $\Sigma_{C,V}$ and $\Sigma_{V,B}$ given in the proof of Theorem 7.3.1, respectively.

We show that in the reduction (Z, T_c) is a certain region for (Σ, D_m) iff the 3SAT instance ϕ is *not* satisfiable.

First assume that (Z, T_c) is a certain region for (Σ, D_m). It is easy to prove that ϕ is *not* satisfiable by contradictions. Conversely, assume that ϕ is *not* satisfiable. That is, for any tuples t_1 and t_2 of R, if $t_1[AX_1 \ldots X_m] = t_2[AX_1 \ldots X_m]$, if $t_1 \rightarrow_{((Z,T_c),\Sigma,D_m)} t_1'$ and $t_2 \rightarrow_{((Z,T_c),\Sigma,D_m)} t_2'$, then $t_1' = t_2'$. It is easy to verify that (Z, T_c) is a certain region for

(Σ, D_m) since $t_1'[C_1 \ldots C_n VB] = t_2'[C_1 \ldots C_n VB]$. □

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 T_c, and if so, how large T_c is.

The *Z-validating* problem is to decide, given (Σ, D_m) and a list Z of attributes, whether there exists a nonempty tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m). The *Z-counting* problem is to determine, given (Σ, D_m) and Z, how many pattern tuples can be found from (Σ, D_m) to construct T_c such that (Z, T_c) is a certain region. Both problems are beyond reach in practice. In particular, the Z-counting problem is as hard as finding the number of truth assignments that satisfy a given 3SAT instance [GJ79].

Theorem 7.3.3: *(1) The Z-validating problem is* NP*-complete. (2) The Z-counting problem is* #P*-complete.* □

Proof: (I) The Z-validating problem is NP-complete.
(1) We show the problem is in NP, by providing an NP algorithm that, given Z, returns 'yes' iff there exists a non-empty pattern tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m). Observe that if so, there exists a tuple t_c consisting of only constants such that $(Z, \{t_c\})$ is a certain region for (Σ, D_m). Thus, it suffices to consider pattern tuples consisting of constants only.

Define active domain dom and variable v as in the proof of Theorem 7.3.1. The NP algorithm works as follows.
(a) Guess a tuple t_c such that for each attribute $A \in Z$, $t_c[A]$ is either a constant in dom or the variable v.
(b) If $(Z, \{t_c\})$ is a certain region for (Σ, D_m), then the algorithm returns 'yes'; and returns 'no', otherwise.

Similar to the proof of Theorem 7.3.1, it is easy to see that the algorithm is in NP and is correct.
(2) We show the problem is NP-hard by reduction from 3SAT. Given an instance ϕ of 3SAT, we construct schemas R and R_m, a master relation D_m of R_m, a set Z of attributes of R, and a set Σ of eRs such that Z is valid iff ϕ is satisfiable.
(a) We define $R(A_1, \ldots, A_m, C_1, \ldots, C_n, V_1, \ldots, V_n)$ such that all attributes have an integer domain. Intuitively, for each $i \in [1, m]$, attribute A_i corresponds to the variable x_i in ϕ, and for each $j \in [1, n]$, C_j and V_j denote the index j and the truth value of the clause C_j , respectively.
(b) The $Z = A_1 \ldots A_m C_1 \ldots C_n$.
(c) We define $R_m(B_1, B_2, B_3, C, V)$ such that all attribute have an integer domain, and

the D_m consists of $7n$ master tuples. For each clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3} (j \in [1,n])$, we define seven master tuples corresponding to the seven truth assignments $(\xi_{j,1}, \ldots, \xi_{j,7})$ that make C_j true:

	B_1	B_2	B_3	C	V
$t_{m_{1,1}}$:	$\xi_{1,1}(x_{p_{11}})$	$\xi_{1,1}(x_{p_{12}})$	$\xi_{1,1}(x_{p_{13}})$	1	1
...	...				
$t_{m_{1,7}}$:	$\xi_{1,7}(x_{p_{11}})$	$\xi_{1,7}(x_{p_{12}})$	$\xi_{1,7}(x_{p_{13}})$	1	1
	...				
$t_{m_{n,1}}$:	$\xi_{n,1}(x_{p_{n1}})$	$\xi_{n,1}(x_{p_{n2}})$	$\xi_{n,1}(x_{p_{n3}})$	n	1
...	...				
$t_{m_{n,7}}$:	$\xi_{n,7}(x_{p_{n1}})$	$\xi_{n,7}(x_{p_{n2}})$	$\xi_{n,7}(x_{p_{n3}})$	n	1

(d) We define $\Sigma = \{\varphi_1, \ldots, \varphi_n\}$. For each clause $C_j = y_{j_1} \vee y_{j_2} \vee y_{j_3}$ $(j \in [1,n])$, we define $\varphi_j = ((A_{p_{j1}} A_{p_{j2}} A_{p_{j3}} C_j, B_1 B_2 B_3 C) \rightarrow (V_j, C), t_{p_j}[C_j] = (j))$.

It is easy to verify that the construction above is a reduction such that Z is valid iff ϕ is satisfiable.

(II) The Z-counting problem is #P-complete.

The reduction above is *parsimonious*. For all pattern tuples t_c, if $(Z, \{t_c\})$ is a certain region for (Σ, D_m), then t_c must satisfy the following: (a) for all attributes C_j $(j \in [1,n])$, $t_c[C_j] = j$; and (b) for all attributes A_i $(i \in [1,m])$, $t_c[A_j]$ must be a constant in $\{0,1\}$. That is, the number of satisfiable truth assignments for the 3SAT instance is equal to the number of pattern tuples t_c such that $(Z, \{t_c\})$ is a certain region for (Σ, D_m). The #3SAT problem, which is the counting version of the 3SAT problem, is #P-complete [GJ79, Val79]. From this it follows that the Z-counting problem is also #P-complete. □

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 set Z of attributes such that (a) $|Z| \leq K$ and (b) there exists a pattern tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m).

This problem is also intractable. Worse still, there exists no approximate algorithm for it with a reasonable bound.

Theorem 7.3.4: *The Z-*minimum *problem is (1)* NP*-complete, and (2) it cannot be approximated within* $c \log n$ *in* PTIME *for a constant c unless* NP $\subseteq$DTIME $(n^{polylog(n)})$.

□

Proof: (I) The *Z-minimum* problem is NP-complete.

(1) We show the problem is in NP by giving an NP algorithm. Consider a set Σ of eRs over schemas (R, R_m), and a positive integer $K \leq |R|$. The algorithm works as follows.
(a) Guess a set Z of attributes in R such that $|Z| \leq K$.
(b) Guess a pattern tuple t_c, and check whether $(Z, t_c[Z])$ is a certain region for (Σ, D_m).
(c) If so, it returns 'yes'; and returns 'no' otherwise.

The correctness of the NP algorithm can be verified along the same lines as the proof of Theorem 7.3.3.
(2) We show that the problem is NP-hard by reduction from the minimum key problem, which is NP-complete [AB96].
(II) The *Z-minimum* problem cannot be approximated within a factor of $c \log n$ in polynomial time for any constant c unless NP $\subseteq$ DTIME $(n^{polylog(n)})$.

This can be verified by an approximation preserving reduction [Vaz03] from the minimum set cover problem, along the same lines as the one for the minimum key problem for functional dependencies (FDs) [AB96]. □

Tractable cases. The intractability results suggest that we consider special cases that allow efficient reasoning.

Fixed Σ. One case is where the set Σ is fixed. Indeed, editing rules are often predefined and fixed in practice.

Concrete T_c. Another case is where no pattern tuples in T_c contain wildcard '_' or $\bar{a}$, *i.e.,* they contain a only.

Direct fix. We also consider a setting in which (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), *i.e.,* $t_{i-1} \rightarrow_{((Z,T_c),\varphi_i,t_{m_i})} t_i$, without extending (Z, T_c).

Each of these restrictions makes our lives much easier.

Theorem 7.3.5: *The consistency problem and the coverage problem are in* PTIME *if we consider (a) a fixed set* Σ *of eRs, (b) a concrete pattern tableau* T_c*, or (c) direct fixes.* □

Proof: Consider (Z, T_c) and (Σ, D_m). Assume *w.l.o.g.* that there is only a single tuple $t_c \in T_c$. When there are multi-tuples in T_c, we can test them one by one by the same algorithms below.

Statements (a) and (b). We first show that if the consistency problem (resp. the coverage problem) is in PTIME for case (b), then it is in PTIME for case (a). We then show that both problems are in PTIME for case (b).

(I) We first show that it suffices to consider case (b) only.

We define active domain dom and variable v as in the proof of Theorem 7.3.1. It suffices to consider R tuples t such that for each attribute A of R, $t[A]$ is either a constant in dom or the variable v.

Since we only need to consider attributes that appear in Σ, there are at most $O(|\mathsf{dom}|^{|\Sigma|})$ tuples of R to be considered, a polynomial when fixing Σ.

For such an R tuple t, if $t \neq t_c \in T_c$, there exist a unique fix, but no certain fix, for t. If $t \rightleftharpoons t_c \in T_c$, it is easy to see that there is a unique fix (resp. certain fix) for t by (Σ, D_m) *w.r.t.* $(Z, \{t_c\})$ if and only if (Σ, D_m) is consistent relative to $(Z, \{t[Z]\})$ (resp. $(Z, \{t[Z]\})$ is a certain region for (Σ, D_m)). From this it follows that we only need to consider case (b).

(II) We show that the consistency problem for case (b) is in PTIME, by giving a PTIME algorithm such that (Σ, D_m) is consistent relative to (Z, T_c) iff the algorithm returns 'yes'. Below we first present algorithm. We then show that the algorithm runs in PTIME. Finally, we verify its correctness.

(1) The algorithm works as follows.

(a) It creates an R tuple t such that $t[Z] = t_c$ and $t[R \setminus Z] = (\_, \ldots, \_)$. It sets $Z_b = Z$, and for each attribute $A \in R$, lets $\mathsf{dep}(A) = \{Z_b\}$ if $A \in Z_b$, and $\mathsf{dep}(A) = \emptyset$ otherwise.

(b) Let $S = \{(\varphi_1, t_{m_1}), \ldots, (\varphi_k, t_{m_k})\}$ be the set of all rule-tuple pairs, where for each $i \in [1,k]$, $(b.1)$ the tuple t_{m_i} is in D_m; $(b.2)$ the eR φ_i is in Σ such that $(\mathsf{LHS}(\varphi_i) \cup \mathsf{LHS}_p(\varphi_i)) \subseteq Z$ and $\mathsf{RHS}(\varphi_i) \notin Z$; $(b.3)$ $t[\mathsf{LHS}(\varphi_i)] = t_{m_i}[\mathsf{LHS}_m(\varphi_i)]$, and tuple t matches the pattern tuple in φ_i.

(c) The algorithm returns 'yes' if the set S is empty, *i.e.,* the tuple t reaches a fix point. Otherwise it does the following.

(d) It checks whether there exist $i, j \in [1,k]$ such that $\mathsf{RHS}(\varphi_i) = \mathsf{RHS}(\varphi_j)$ and $t_{m_i}[\mathsf{RHS}_m(\varphi_i)] \neq t_{m_j}[\mathsf{RHS}_m(\varphi_j)]$. If so, it returns 'no'. Otherwise it does the following.

(e) For each $i \in [1,k]$, let $\mathsf{dep}(\mathsf{RHS}(\varphi_i)) = \mathsf{dep}(\mathsf{RHS}(\varphi_i)) \cup \{\mathsf{LHS}(\varphi_i)\}$, $t[\mathsf{RHS}(\varphi_i)] = t_{m_i}[\mathsf{RHS}_m(\varphi_i)]$, and $Z = Z \cup \{\mathsf{RHS}(\varphi_1), \ldots, \mathsf{RHS}(\varphi_k)\}$.

(f) The algorithm then checks whether there exists an eR φ in Σ such that $\mathsf{LHS}(\varphi) \subseteq Z$ and $\mathsf{RHS}(\varphi) \in Z \setminus Z_b$; and there is a master tuple t_m in D_m that can be applied to tuple t; and moreover, they satisfy the following conditions:

- For each attribute $A \in \mathsf{LHS}(\varphi)$, there exists a non-empty $X \in \mathsf{dep}(A)$ such that $\mathsf{RHS}(\varphi) \notin X$; and
- $t_m[\mathsf{RHS}_m(\varphi)] \neq t[\mathsf{RHS}(\varphi)]$.

If so, the algorithm returns 'no'.

(g) Otherwise, the algorithm repeats the steps (b), (c), (d), (e), and (f) above.

(2) To see that the algorithm is in PTIME, observe that (a) each time Z is expanded by at least one more attribute, and (b) there are $|\Sigma| \times |D_m|$ rule-tuple pairs, and once such a pair is applied to the tuple t, it will not be considered again. As the number of tuples of R is a polynomial, the process above can be done in polynomial time.

(3) We next show the correctness of the algorithm.

First assume that the algorithm returns 'yes'. We show that (Σ, D_m) is consistent relative to (Z, T_c), by contradiction. Assume that (Σ, D_m) is not consistent relative to (Z, T_c). Then there exist two distinct sequences $T_1 = t \to t_1 \to \ldots \to t_k$ and $T_2 = t \to s_1 \to \ldots \to s_h$ such that $k, h \leq |R|$ and $t_k \neq s_h$. Let $B \in R$ be the first attribute such that (a) B is updated in both t_i and s_j such that $t_i[B] \neq s_j[B]$. There are two cases : (a) $i = j$ and (b) $i \neq j$. In case (a), the algorithm will update all attributes to be updated before t_i is in T_1 and s_j in T_2. When updating the attribute B, the conflict can be detected at step (d), and the algorithm would return 'no'. In case (b), the algorithm will update all attributes to be updated before t_{j+1} is in T_1 and s_{j+1} in T_2 if $j < i$ (similarly for $j > i$). Then when updating B as for t_i, the conflict can be detected at step (f), and the algorithm would return 'no'.

Conversely, assume that the algorithm returns 'no'. We show that (Σ, D_m) is not consistent relative to (Z, T_c). There are two cases : (a) the algorithm returns 'no' at step (d), and (b) it returns 'no' at step (g). In case (a) it is easy to see that (Σ, D_m) is not consistent relative to (Z, T_c). In case (b), the update of the attribute $\mathsf{RHS}(\varphi)$ can be delayed such that the case for step (d) will appear. Thus, (Σ, D_m) is not consistent relative to (Z, T_c) either.

(III) We next show that the coverage problem for case (b) is in PTIME. Indeed, the PTIME algorithm developed above can be applied here, but it only returns 'yes' at step (c) if when the set S is empty and the tuple t only consists of constants.

This completes the proof for statements (a) and (b).

Statement (c). We show that the consistency and coverage problems are in PTIME for direct fixes, one by one as follows.

(I) We first show how to check the relative consistency via SQL queries, which yields a PTIME algorithm for the problem.

Given a set Z of certain attributes, let Σ_Z be the set of eRs φ in Σ such that $\mathsf{LHS}(R, \varphi) \subseteq Z$, but $\mathsf{RHS}(R, \varphi) \notin Z$.

We first define an SQL query Q_φ for an eR $\varphi = ((X, X_m) \to (B, B_m), t_p[X_p])$ in Σ_Z, as follows.

1. **select distinct** (X_m, B_m) **as** (X, B)
2. **from** $\mathsf{R_m}$
3. **where** $\mathsf{R_m}.\mathsf{X_p} \approx t_p[X_p]$ **and** $\mathsf{R_m}.\mathsf{X_m} \rightleftharpoons t_c[X]$

We use $Q_\varphi(X, B)$ and $Q_\varphi(X)$ to denote the result of the SQL query projected on attributes $X \cup \{B\}$ and X, respectively.

We then define an SQL query $Q_{(\varphi_1,\varphi_2)}$ for two eRs $\varphi_1 = ((X_1X, X_{m_1}X_m) \to (B, B_{m_1}), t_{p_1}[X_{p_1}])$ and $\varphi_2 = ((X_2X, X_{m_2}X'_m) \to (B, B_{m_2}), t_{p_2}[X_{p_2}])$ such that $X_1 \cap X_2$ is empty and $|X| = |X_m| = |X'_m|$. Here X may be empty.

1. **select** $R_1.X_1$,$R_1.X$,$R_2.X_2$
2. **from** $Q_{\varphi_1}(X_1X, B)$ **as** R_1, $Q_{\varphi_2}(X_2X, B)$ **as** R_2
3. **where** $R_1.X = R_2.X$ **and** $R_1.B \neq R_2.B$

For two eRs $\varphi_1 = ((X_1, X_{m_1}) \to (B_1, B_{m_1}), t_{p_1}[X_{p_1}])$ and $\varphi_2 = ((X_2, X_{m_2}) \to (B_2, B_{m_2}), t_{p_2}[X_{p_2}])$ such that $B_1 \neq B_2$, we define SQL query $Q_{(\varphi_1,\varphi_2)}$ that always returns $\emptyset$.

Observe that that (Σ, D_m) is consistent relative to $(Z, \{t_c\})$ if and only if for all eRs φ_1 and φ_2 in Σ_Z, the queries $Q_{(\varphi_1,\varphi_2)}$ return an empty result. The SQL query $Q_{(\varphi_1,\varphi_2)}$ is obviously in PTIME, and hence, the consistency problem is in PTIME for direct fixes.
(II) For the coverage problem, observe that (Z, T_c) is a certain region for (Σ, D_m) if and only if:
(a) (Σ, D_m) is consistent relative to (Z, T_c).
(b) for each $B \in R \setminus Z$, there exists an eR $\varphi = ((X, X_m) \to (B, B_m), t_p[X_p])$ in Σ such that (a) $X \in Z$ and $t_c[X]$ consists of only constants, (b) $t_p[X_p] \approx t_c[X_p]$, and (c) there is a master tuple $t_m \in D_m$ with $t_m[X_m] = t_c[X]$.

Both conditions are checkable in PTIME, and hence, so is the coverage problem. □

However, it does not simplify the other problems.

Corollary 7.3.6: *When only direct fixes are considered, the Z-validating, Z-counting and Z-minimum problems remain* NP*-complete,* #P*-complete, both* NP*-complete and approximation-hard, respectively.* □

Proof: (1) For the Z-validating problem and the Z-counting problem, by a careful checking of the proofs in Theorems 7.3.3, it shows that the proofs in Theorem 7.3.3 work for this special case.

(2) The NP-hardness of the Z-minimum problem is shown by reduction from the minimum node cover problem, which is NP-complete [GJ79].

A node cover in a graph $G(V,E)$ is a subset $V' \subseteq V$ such that for each edge (u,v) of the graph, at least one of u and v belongs to the set V'. Given a graph $G(V,E)$ and a positive integer $K \leq |V|$, the node cover problem asks whether there exists a node cover V' in G having $|V'| \leq K$.

Consider an instance VC = $(G(V,E),K)$ of the node cover problem, where $V = \{v_1,\ldots,v_{|V|}\}$ and $E = \{e_1,\ldots,e_{|E|}\}$. We construct schemas (R,R_m) and a set Σ of eRs such that there is a solution to VC if and only if there is a solution to the constructed minimum Z instance.

(a) We define $R(A_1, \ldots, A_{|V|}) = R_m(B_1, \ldots, B_{|V|})$. Intuitively, for each $i \in [1,|V|]$, the R attribute A_i and the R_m attribute B_i correspond to the node v_i in V.

(b) The master relation D_m consists of a single master tuple $t_m = (1,\ldots,1)$.

(c) The Σ consists of $2 * |E|$ eRs. For each $e = (v_i,v_j)$ $(1 \leq i,j \leq |V|)$, we define two eRs $\varphi_{e,1}$ and $\varphi_{e,2}$, where $\varphi_{e,1} = ((A_i,B_i) \rightarrow (A_j,B_j))$, and $\varphi_{e,2} = ((A_j,B_j) \rightarrow (A_i,B_i))$, which have empty pattern tuples.

(d) Let K' = K + the number of isolated nodes in G.

It is easy to verify that there is a node cover V' to VC with $|V'| \leq K$ iff there is a certain region (Z,T_c) with $|Z| \leq K'$.

(3) The approximation hardness of the Z-minimum problem is shown by an approximation preserving reduction [Vaz03] from the minimum set cover problem, a minor modification of the one for the minimum key problem [AB96]. □

One might think that fixing master data D_m would also simplify the analysis of eRs. Unfortunately, it does not help.

Corollary 7.3.7: *Both the consistency problem and the coverage problem remain co*NP*-complete when D_m is fixed.* □

Proof: The reductions given in the proofs of Theorem 7.3.1 and Theorem 7.3.2 both use a *fixed* master relation, which have five attributes and three master tuples. As a result, the coNP lower bounds remain intact for the consistency and coverage problems when the master relation D_m is fixed. □

7.4 Computing Certain Regions

An important issue concerns how to automatically derive a set of certain regions from a set Σ of eRs and a master relation D_m. These regions are recommend to users, such that Σ and D_m warrant to find an input tuple t a certain fix as long as the users assure that t is correct in any of these regions. However, the intractability and approximation-hardness of Theorems 7.3.2, 7.3.3 and 7.3.4 tell us that any efficient algorithms for deriving certain regions are necessarily heuristic.

We develop a heuristic algorithm based on a *characterization* of certain regions as cliques in graphs. Below we first introduce the characterization and then present the algorithm. We focus on direct fixes, a special case identified in Section 7.3.2 that is relatively easier (Theorem 7.3.5) but remains intractable for deriving certain regions (Corollary 7.3.6).

7.4.1 Capturing Certain Regions as Cliques

We first introduce a notion of compatible graphs to characterize eRs and master data. We then establish the connection between certain regions and cliques in such a graph.

Compatible graphs. Consider $\Sigma = \{ \varphi_i \mid i \in [1,n]\}$ defined on (R, R_m), where $\varphi_i = ((X_i, X_{m_i}) \to (B_i, B_{m_i}), t_{p_i}[X_{p_i}])$. We use the following notations.

(1) For a list X'_i of attributes in X_i, we use $\lambda_{\varphi_i}(X'_i)$ to denote the corresponding attributes in X_{m_i}. For instance, when $(X_i, X_{m_i}) = (ABC, A_mB_mC_m)$, $\lambda_{\varphi_i}(AC) = A_mC_m$.

We also use the following: (a) $\mathsf{LHS}(\varphi_i) = X_i$, $\mathsf{RHS}(\varphi_i) = B_i$; (b) $\mathsf{LHS}_m(\varphi_i) = X_{m_i}$, $\mathsf{RHS}_m(\varphi) = B_{m_i}$; and (c) $\mathsf{LHS}_p(\varphi_i) = X_{p_i}$. For a set Σ_c of eRs, we denote $\cup_{\varphi\in\Sigma_c}\mathsf{LHS}(\varphi)$ by $\mathsf{LHS}(\Sigma_c)$; similarly for $\mathsf{RHS}(\Sigma_c)$, $\mathsf{LHS}_m(\Sigma_c)$ and $\mathsf{RHS}_m(\Sigma_c)$.

(2) Consider pairs (φ_i, t_m) and (φ_j, t'_m) of eRs and master tuples such that $t_{p_i}[X_{p_i}] \approx t_m[\lambda_{\varphi_i}[X_{p_i}]]$ and $t_{p_j}[X_{p_j}] \approx t'_m[\lambda_{\varphi_j}[X_{p_j}]]$. We say that t_m and t'_m are *conflict tuples* if (a) $B_i = B_j$ and $t_m[B_{m_i}] \neq t'_m[B_{m_j}]$, and (b) for each attribute $A \in X_i \cap X_j$, $t_m[\lambda_{\varphi_i}(A)] = t'_m[\lambda_{\varphi_j}(A)]$.

That is, (φ_i, t_m) and (φ_j, t'_m) may incur conflicts when they are applied to the same input tuple. To avoid taking conflict tuples in T_c, we remove conflict tuples from D_m, and refer to the result as the *reduced* master data D_s.

(3) We say that eR-tuple pairs (φ_i, t_m) and (φ_j, t'_m) are *compatible* if (a) $B_i \neq B_j$, $B_i \notin X_j$, $B_j \notin X_i$, and (b) for each attribute $A \in X_i \cap X_j$, $t_m[\lambda_{\varphi_i}(A)] = t'_m[\lambda_{\varphi_j}(A)]$. Intuitively, we can apply (φ_i, t_m) and (φ_j, t'_m) to the same input tuple t if they are compatible.

We are now ready to define compatible graphs.

The *compatible graph* $G(V,E)$ of (Σ, D_m) is an undirected graph, where (1) the set V of nodes consists of eR-tuple pairs (φ_i, t_m) such that $\varphi_i \in \Sigma$, $t_m \in D_s$, and $t_{p_i}[X_{p_i}] \approx t_m[\lambda_{\varphi_i}[X_{p_i}]]$; and (2) the set E of edges consists of (u,v) such that u and v in V are compatible with each other.

The graph $G(V,E)$ depicts what eR-tuple pairs are compatible and can be applied to the same tuple. Note that V (resp. E) is bounded by $O(|\Sigma||D_m|)$ (resp. $O(|\Sigma|^2|D_m|^2)$).

The connection. We now establish the connection between identifying certain regions (Z,T_c) for (Σ,D_m) and finding cliques $\mathcal{C}$ in the compatible graph $G(V,E)$.

Consider a clique $\mathcal{C} = \{v_1,\ldots,v_k\}$ in G, where for each $i \in [1,k]$, $v_i = (\varphi_i, t_{m_{j_i}})$. Let $\Sigma_\mathcal{C}$ be the set of eRs in the clique $\mathcal{C}$. Then it is easy to verify (a) $\mathsf{LHS}(\Sigma_\mathcal{C}) \cap \mathsf{RHS}(\Sigma_\mathcal{C}) = \emptyset$, and (b) $|\mathsf{RHS}(\Sigma_\mathcal{C})| = |\mathcal{C}| = k$, *i.e.,* the number of attributes in $\mathsf{RHS}(\Sigma_\mathcal{C})$ is equal to the number of nodes in $\mathcal{C}$.

Let $Z = R \setminus \mathsf{RHS}(\Sigma_\mathcal{C})$, and t_c be a tuple with attributes in Z such that (a) $t_c[\mathsf{LHS}(\Sigma_\mathcal{C})] = t_{j_1} \bowtie \ldots \bowtie t_{j_k}[\mathsf{LHS}(\Sigma_\mathcal{C})]$, where for each $i \in [1,k]$, $t_{j_i}[X_iB_i] = t_{m_{j_i}}[X_{m_i}B_{m_i}]$, and (b) $t_c[A] =$ '_' for all remaining attributes $A \in Z$. Here $\bowtie$ is the natural join operator. Then it is easy to verify that $(Z,\{t_c\})$ is a certain region for (Σ,D_m). Hence we have:

Proposition 7.4.1: *Each clique in the compatible graph G of* (Σ,D_m) *corresponds to a certain region for* (Σ,D_m). □

Proof: Consider a clique $\mathcal{C} = \{v_1,\ldots,v_k\}$ in G, where for each $i \in [1,k]$, $v_i = (\varphi_i, t_{m_{j_i}})$. Let $\Sigma_\mathcal{C}$ be the set of eRs in the clique $\mathcal{C}$. Following from the definition of the compatible graph, we have the following: (a) $\mathsf{LHS}(\Sigma_\mathcal{C}) \cap \mathsf{RHS}(\Sigma_\mathcal{C}) = \emptyset$; and (b) $|\mathsf{RHS}(\Sigma_\mathcal{C})| = |\mathcal{C}| = k$, *i.e.,* the number of attributes in $\mathsf{RHS}(\Sigma_\mathcal{C})$ is equal to the number of nodes in $\mathcal{C}$.

Let $Z = R \setminus \mathsf{RHS}(\Sigma_\mathcal{C})$, and t_c be a tuple with attributes in Z such that (a) $t_c[\mathsf{LHS}(\Sigma_\mathcal{C})] = t_{j_1} \bowtie \ldots \bowtie t_{j_k}[\mathsf{LHS}(\Sigma_\mathcal{C})]$, where for each $i \in [1,k]$, $t_{j_i}[X_iB_i] = t_{m_{j_i}}[X_{m_i}B_{m_i}]$, and (b) $t_c[A] =$ '_' for all remaining attributes $A \in Z$.

Since there are no conflict tuples in the compatible graph, for all R tuples $t \rightleftharpoons t_c$, the eR-tuple pairs in the clique $\mathcal{C}$ guarantee that the tuple t has a certain fix. Hence, we can derive a certain region for (Σ,D_m) from each clique in the compatible graph G. □

This allows us to find certain regions by employing algorithms (*e.g.,* [JPY88, MU04]) for finding maximal cliques in a graph.

Compressed graphs. However, the algorithms for finding cliques take $O(|V||E|)$ time on each clique. When it comes to compatible graph, it takes $O(|\Sigma|^3|D_m|^3)$ time for

each certain region, too costly to be practical on large D_m.

In light of this we compress a compatible graph $G(V,E)$ by removing master tuples from the nodes. More specifically, we consider *compressed compatible graph* $G^c(V^c,E^c)$, where (1) V^c is Σ, *i.e.,* each node is an eR in Σ, and (2) there is an edge (φ_i,φ_j) in E^c if and only if there exist master tuples t_m,t'_m such that $((\varphi_i,t_m),(\varphi_j,t'_m))$ is an edge in E.

Observe that G^c is much smaller than G and is independent of master data D_m: V^c is bounded by $O(|\Sigma|)$ and E^c is bounded by $O(|\Sigma|^2)$. On the other hand, however, it is no longer easy to determine whether a clique yields a certain region. More specifically, let $\mathcal{C}$ be a clique in G^c and $Z = R \setminus \mathsf{RHS}(\Sigma_{\mathcal{C}})$. The Z-validating problem *for a clique* is to determine whether there exists a nonempty pattern tableau T_c such that (Z,T_c) is a certain region for (Σ,D_m).

Theorem 7.4.2: *The Z-validating problem for a clique in a compressed graph G^c is* NP*-complete.* □

Proof: By reduction from the 3SAT problem to the Z-validating problem, where (a) for each attribute $B \in (R \setminus Z)$, there is exactly one eR φ with $\mathsf{LHS}(\varphi) = B$, and (b) the eRs form a clique in the compressed compatible graph. The reduction in the proof of Theorem 7.3.3 is such a reduction. From this, it follows that the Z-validating problem for a clique in a compressed graph G^c is NP-complete. □

A heuristic. To cope with the intractability we develop a heuristic algorithm to validate Z. We partition Z into Z_1 and Z_2 such that only Z_2 is required to match a list Z_m of attributes in R_m, where the correctness of Z_1 is to be assured by the users. Here Z_2 and Z_m are $\mathsf{LHS}(\Sigma_{\mathcal{C}})$ and $\mathsf{LHS}_m(\Sigma_{\mathcal{C}})$, respectively, derived from a clique $\mathcal{C}$ in the compressed graph G^c. We denote this by $W = (Z_1,Z_2 \parallel Z_m)$, where $Z = Z_1 \cup Z_2$, $Z_1 \cap Z_2 = \emptyset$ and $|Z_2| = |Z_m|$.

The W-validating problem asks whether there exists t_m in D_s such that $(Z_1Z_2,\{t_c\})$ is a certain region for (Σ,D_s), where $t_c[Z_1]$ consists of '_' only and $t_c[Z_2] = t_m[Z_m]$. That is, t_c is extracted from a single master tuple, not a combination from multiple. In contrast to Theorem 7.4.2, we have:

Proposition 7.4.3: *There exists an $O(|\Sigma||D_s|\log|D_s|)$-time algorithm for the W-validating problem.* □

Proof: We show this by giving algorithm validateW, shown in Fig. 7.2. We first show that the algorithm runs in $O(|\Sigma||D_s|\log|D_s|)$ time, and then verify its correctness.
(I) We first show the algorithm is in $O(|\Sigma||D_s|\log|D_s|)$ time.

Algorithm validateW

Input: W = $(Z_1, Z_2 \parallel Z_m)$, a set Σ of eRs on schemas (R, R_m), and a reduced master relation D_s of R_m.

Output: true if W is valid, or false otherwise.

1. $t := \emptyset$; /* t is an R tuple */
2. **for** each master tuple t_m in D_s **do**
3. $t[Z_2] := t_m[Z_m]$; $t[R \setminus Z_2] := (\_, \ldots, \_)$;
4. **for** each φ in Σ having $X \subseteq Z_2$ and $B \notin (Z_1 Z_2)$ **do**
   /* Here $\varphi = ((X, X_m) \to (B, B_m), t_p[X_p])$ */
5. **if** $t[X_p] \approx t_p[X_p]$ **and** $t'_m[X_m] = t[X]$ $(t'_m \in D_s)$
6. **then** $t[B] := t'_m[B_m]$;
7. **if** $t[R \setminus Z_1]$ contains only constants **then return** true;
8. **return** false.

Figure 7.2: Algorithm validateW

- For each eR $\varphi \in \Sigma$ such that $\mathsf{LHS}(\varphi) \in Z_2$ and $\mathsf{RHS}(\varphi) \notin (Z_1 Z_2)$, we first build a *hash* index based on $\mathsf{LHS}_m(\varphi)$ for master tuples in D_s. This takes $O(|\Sigma||D_s| \log |D_s|)$ time, and this part is not shown in the pseudo-code.
- There are at most $O(|D_s||\Sigma|)$ loops (lines 2–7). And each innermost loop takes $O(\log |D_s|)$ time (lines 5–6) This takes $O(|\Sigma||D_s| \log |D_s|)$ time.

Putting these together, it is easy to see that the algorithm indeed runs in $O(|\Sigma||D_s| \log |D_s|)$ time.

(II) We now show the correctness of the algorithm.

That is, given $W = (Z_1, Z_2 \parallel Z_m)$, there is a non-empty pattern tableau T_c such that $(Z_1 Z_2, T_c)$ is a certain region for (Σ, D_m) iff algorithm validateW returns true. Recall that we only focus on direct fixes (Section 7.3).

First, assume that the algorithm returns 'yes'. Then there exists a master tuple t_m that makes $t[R \setminus Z_1]$ contain only constants. It suffices to show that $(Z, t_m[Z_m])$ is a certain region. Indeed, this is because (a) the algorithm guarantees that for all tuples t of R, if $t[Z] = t_m[Z_m]$, then $t[A]$ is a constant for all attributes $A \in (R \setminus Z_1 Z_2)$ (line 6), and (b) there are no conflict tuples in D_s.

Conversely, assume that W is valid. That is, there must exist a master tuple t_m in D_s such that $(Z_1 Z_2, \{t_c\})$ is a certain region for (Σ, D_s), where $t_c[Z_1]$ consists of '_' only and $t_c[Z_2] = t_m[Z_m]$.

For the master tuple t_m, $t[R \setminus Z_1]$ must contain only constants, and the algorithm

Algorithm compCRegions

Input: A number K, a set Σ of eRs defined on (R, R_m), and a master relation D_m of R_m.

Output: An array M of regions (Z, T_c).

1. Compute D_s from D_m by removing conflict tuples;
2. Build the compressed compatible graph G^c for (Σ, D_s);
3. $M := \emptyset$; $\Gamma :=$ findCliques(K, G^c);
4. **for** each clique $\mathcal{C}$ in Γ **do**
5. $S :=$ cvrtClique $(\mathcal{C}, \Sigma, D_s)$;
6. **for** each (Z, t_c) in S **do**
7. $M[Z] := (Z, M[Z].T_c \cup \{t_c\})$;
8. **return** M.

Procedure cvrtClique

Input: A clique $\mathcal{C}$ in the graph G^c, a set Σ of eRs on schemas (R, R_m), and a reduced master relation D_s of R_m.

Output: A set S of (Z, t_c) pairs.

1. $S := \emptyset$; $Z_2 := \mathsf{LHS}(\Sigma_{\mathcal{C}})$; $Z_m := \mathsf{LHS}_m(\Sigma_{\mathcal{C}})$;
2. **for** each master tuple t_m in D_s **do**
3. $t[Z_2] := t_m[Z_m]$; $t[R \setminus Z_2] := (\_, \ldots, \_)$; /* t is an R tuple */
4. **for** each φ in Σ with $X \subseteq Z_2$ and $B \in \mathsf{RHS}(\Sigma_{\mathcal{C}})$ **do**
   /* Here $\varphi = ((X, X_m) \to (B, B_m), t_p[X_p])$ */
5. **if** $t[X_p] \approx t_p[X_p]$ **and** $t'_m[X_m] = t[X]$ $(t'_m \in D_s)$
6. **then** $t[B] := t'_m[B_m]$;
7. $S := S \cup \{(Z_1 Z_2, t[Z_1 Z_2])\}$; /* $t[Z_1]$ contains exactly '_' */
8. **return** S;

Figure 7.3: A Graph-based Algorithm

returns true. □

Based on this, the algorithm works as follows. Given Z and a clique $\mathcal{C}$, it first partitions Z into $W = (Z_1, Z_2 \parallel Z_m)$. It then finds a tuple t_c using the $O(|\Sigma||D_s| \log |D_s|)$-time algorithm. To ensure the correctness we require that for any φ_i and φ_j in G^c, $\mathsf{LHS}(\varphi_i)$ and $\mathsf{LHS}(\varphi_j)$ are disjoint. In fact, a set of certain regions can be generated when validating W, one for each master tuple in D_s. We employ this idea to generate certain regions from cliques in the compressed compatible graph (in Procedure cvrtClique).

7.4.2 A Graph-based Heuristic Algorithm

Based on the graph characterization we presented Algorithm compCRegions in Fig. 7.3. It first computes a *reduced* master relation D_s (line 1), and builds the compressed compatible graph G^c of (Σ, D_s) (line 2). It then invokes Procedure findCliques to find up to K maximal cliques in G^c (line 3). These cliques are converted into certain regions by Procedure cvrtClique (line 5). Finally, it constructs M by merging certain regions with the same Z (lines 6-7). Here M is guaranteed nonempty since (a) every graph has at least one maximal clique, and (b) cvrtClique finds a certain region from each clique (see below).

Procedure findCliques is presented following the algorithm given in [JPY88] for the ease of understanding. However, we used the algorithm in [MU04] in the experiments. These algorithms output a maximal clique in $O(|V||E|)$ time for a graph $G(V,E)$, in a lexicographical order of the nodes. Procedure findCliques first generates a total order for eRs in Σ (lines 1–3). Then it recursively generates K maximal cliques (lines 4–12). This part is a simulation of the algorithm in [JPY88], which outputs maximal independent sets. This takes $O(K|\Sigma|^3)$ time in total. The correctness of Procedure findCliques is ensured by that of the algorithm in [JPY88].

Procedure findCliques makes use of the methods of [JPY88, MU04] to find maximal cliques. Those methods have proven effective and efficient in practice. Indeed, the algorithm of [MU04] can find about 100,000 maximal cliques per second on sparse graphs (http://research.nii.ac.jp/∼uno/code/mace.htm).

Given a clique $\mathcal{C}$, Procedure cvrtClique derives a set of certain regions, using the heuristic given in Section 7.4.1. It first extracts Z_2 and Z_m from the set $\Sigma_{\mathcal{C}}$ of eRs in $\mathcal{C}$ (line 1). For each master tuple t_m, it then identifies a set Z_1Z_2 of attributes and a pattern $t[Z_1Z_2]$ such that $(Z_1Z_2, \{t[Z_1Z_2]\})$ forms a certain region (lines 2–7). The rational behind this includes: (a) no conflict tuples are in D_s, and (b) for any $B \in (R \setminus Z_2)$, $t[B]$ is a constant taken from D_s (line 7).

Example 7.4.1: Consider the master relation D_m in Fig. 7.1(b), and $\Sigma' = \{\varphi_{(\mathsf{FN},2)}, \varphi_{(\mathsf{LN},2)}, \varphi_{(\mathsf{AC},1)}, \varphi_{(\mathsf{str},1)}, \varphi_{(\mathsf{city},1)}, \varphi_4\}$ consisting of eRs derived from φ_1, φ_2 and φ_4 of Example 7.2.1 by, *e.g.,* instantiating B_1 with AC, str and city.

The algorithm first builds a compressed graph $G^c(V^c, E^c)$ such that $V^c = \Sigma'$, and there is an edge in E^c for all node pairs except for two node pairs $(\varphi_{(\mathsf{AC},1)}, \varphi_4)$ and $(\varphi_{(\mathsf{city},1)}, \varphi_4)$. For $K = 2$, findCliques finds two cliques $\mathcal{C}_1 = \{\varphi_{(\mathsf{FN},2)}, \varphi_{(\mathsf{LN},2)}, \varphi_{(\mathsf{AC},1)}, \varphi_{(\mathsf{str},1)}, \varphi_{(\mathsf{city},1)}\}$ and $\mathcal{C}_2 = \{\varphi_{(\mathsf{FN},2)}, \varphi_{(\mathsf{LN},2)}, \varphi_{(\mathsf{str},1)}, \varphi_4\}$, by checking eRs following

their order in Σ'.

The algorithm returns two certain regions (Z_1, T_{c_1}) and (Z_2, T_{c_2}), where $Z_1 = (\mathsf{zip}, \mathsf{phn}, \mathsf{type}, \mathsf{item})$ with $T_{c_1} = \{t_{1,1}, t_{1,2}\}$ and $Z_2 = (\mathsf{zip}, \mathsf{phn}, \mathsf{AC}, \mathsf{type}, \mathsf{item})$ with $T_{c_2} = \{t_{2,1}, t_{2,2}\}$. For each $i \in [1,2]$, (a) $t_{1,i}[\mathsf{type}, \mathsf{item}] = (\_, \_)$, $t_{1,i}[\mathsf{zip}, \mathsf{phn}] = s_i[\mathsf{zip}, \mathsf{Mphn}]$; and (b) $t_{2,i}[\mathsf{type}, \mathsf{item}] = (\_, \_)$, $t_{2,i}[\mathsf{zip}, \mathsf{phn}, \mathsf{AC}] = s_i[\mathsf{zip}, \mathsf{Mphn}, \mathsf{AC}]$ for s_i of Fig. 7.1(b). □

A preference model. When there exist more than K maximum cliques we need to decide which K cliques to pick. To this end, the algorithm adopts a preference model that ranks certain regions (Z, T_c) based on the following factors.

- The *number* $|Z|$ *of attributes* in Z. We naturally want Z to be as small as possible. The larger the size of a clique $\mathcal{C}$ is, the smaller $|Z|$ is for Z derived from $\mathcal{C}$.
- The *accuracy* $\mathsf{ac}(A)$ of $A \in R$, indicating the confidence placed by the user in the accuracy of the attribute. The smaller $\mathsf{ac}(A)$ is, the more reliable A is.

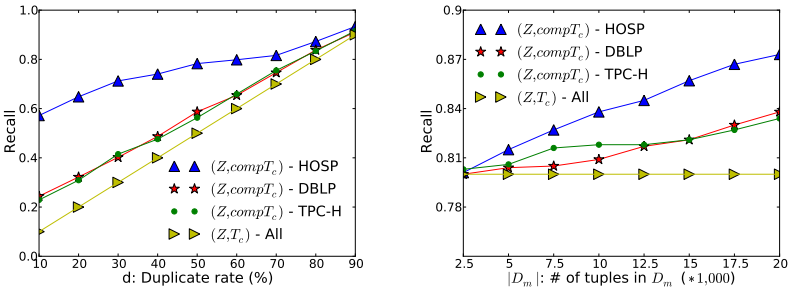
The algorithm uses a total order $\mathcal{O}$ for the eRs in Σ such that $\mathcal{O}(\varphi) < \mathcal{O}(\varphi')$ if $\mathsf{ac}(\mathsf{RHS}(\varphi)) < \mathsf{ac}(\mathsf{RHS}(\varphi'))$. It finds maximum cliques (small regions). Cliques having eRs φ with unreliable $\mathsf{RHS}(\varphi)$ are returned first. Hence, small Z with reliable attributes derived from the cliques are selected.

Correctness and complexity. The algorithm guarantees to return a nonempty set M of certain regions, by Propositions 7.4.1 and 7.4.3. It is in $O(|\Sigma|^2|D_m|\log|D_m| + K|\Sigma|^3 + K|\Sigma||D_m|\log|D_m|)$ time: it takes $O(|\Sigma|^2|D_m|\log|D_m|)$ time to build a compressed compatible graph (lines 1-2), $O(K|\Sigma|^3)$ time to find cliques (line 3), and $O(K|\Sigma||D_m|\log|D_m|)$ time to derive certain regions from the cliques (lines 4-7). In practice, $|\Sigma|$ and K are often small. We verify its effectiveness and efficiency in Section 7.5.

7.5 Experimental Study

We next present an experimental study using both real-life data and synthetic data. Two sets of experiments were conducted to verify (1) the effectiveness of the certain regions obtained; and (2) the efficiency and scalability of algorithm compCRegions in deriving certain regions. For the effectiveness study, we used the incremental repairing algorithm developed in [CFG+07], IncRep, for comparison.

Experimental setting. Real-life data (HOSP and DBLP) was used to test the efficacy of certain regions derived by our algorithm in real world. Synthetic data (TPC-H and RAND) was employed to control the characteristics of data and editing rules, for an in-depth analysis.



(a) Varying $d\%$ (b) Varying $|D_m|$

Figure 7.4: Tuple Level Recall

(1) HOSP *(Hospital Compare) data* is publicly available from U.S. Department of Health & Human Services[1]. There are 37 eRs designed for HOSP.

(2) DBLP *data* is from the DBLP Bibliography[2]. There are 16 eRs designed for DBLP.

(3) TPC-H *data* is from the DBGEN generator [3]. There are 55 eRs designed for TPC-H.

(4) RAND *data* was generated by varying the following parameters: (a) the number of attributes in the master relation R_m; (b) the number of attributes in the relation R; (c) the number of master tuples; (d) the number of editing rules (eRs); and (e) the number of attributes in LHS of eRs.

We refer the reader to the appendix for the details of the datasets, the editing rules designed, and Algorithm IncRep.

Implementation. All algorithms were implemented in Python 2.6, except that the *C* implementation[4] in [MU04] was used to compute maximal cliques. All experiments were run on a machine with an Intel Core2 Duo P8700 (2.53GHz) CPU and 4GB of memory. Each experiment was repeated over 5 times and the average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness. We used real-life datasets HOSP and DBLP, and synthetic TPC-H data to verify the effectiveness of certain regions found by our heuristic compCRegions.

The tests were conducted upon varying three parameters: $d\%$, $|D_m|$ and $n\%$, where $d\%$ means the probability that an input tuple can match a tuple in D_m; $|D_m|$ is the cardinality of master data; $n\%$ is the noise rate, which represents the percentage of attributes with errors in the input tuples. When one parameter was varied, all the others were fixed.

The comparisons were quantified with two measures, in tuple level and in attribute

[1] http://www.hospitalcompare.hhs.gov/
[2] http://www.informatik.uni-trier.de/~ley/db/
[3] http://www.tpc.org/tpch/
[4] http://research.nii.ac.jp/~uno/code/mace.htm

level, respectively.

Tuple level comparison. The tuple level recall is defined as:

$$recall_t = \#\text{ of corrected tuples} \;/\; \#\text{ of error tuples}$$

The results for varying $d\%$ and $|D_m|$ are shown in Fig. 7.4(a) and 7.4(b), respectively. Notably for the (Z, T_c) derived, its recall is close to the duplicate rate $d\%$, irrelevant to the datasets tested. Hence, in Fig. 7.4(a) and Fig. 7.4(b), the curve annotated by (Z, T_c)-ALL is used to represent the curves for all datasets. Moreover, $compT_c$ stands for the complete T_c.

In Fig. 7.4(a), we fixed $|D_m|$ = 20000 while varying $d\%$ from 10% to 90%. When the master data covers more portions of the input tuples (from 10% to 90%), the recall increases (from 0.1 to 0.9). This tells us the following: (1) the efficacy of certain regions is sensitive to duplicate rates. Hence, the master data should be as complete as possible to cover the input tuples; and (2) the effect of certain regions (Z, T_c) derived by compCRegions is worse than complete T_c, as expected, since some valid pattern tuples were not selected by our heuristic method. However, when $d\%$ is increased, the recall via heuristic (Z, T_c) becomes close to that of the complete T_c, validating the effectiveness of compCRegions.

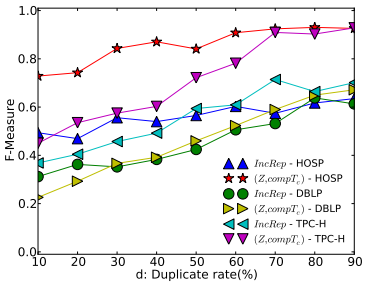
In Fig. 7.4(b), we fixed $d\%$ = 80% while varying $|D_m|$ from 2500 to 20000. The recall of $(Z, compT_c)$ increases when increasing $|D_m|$, as expected. Observe that the curve for HOSP grows faster than the ones for TPC-H or DBLP. This is data-dependent, due to the fact that the number of hospitals in US is much smaller than the distinct entities in TPC-H sale records or DBLP publications. By increasing $|D_m|$, HOSP has a higher probability to cover more portions of the input tuples, reflected in Fig. 7.4(b). This reveals that the completeness of master data is pivot. When D_m is assured consistent and complete [RW08], our algorithm can find certain regions with good recalls.

Attribute level comparison. For the attribute level quantification, we used a fine-grained measure F-measure [FM]:

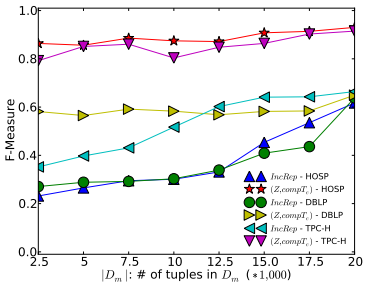
$$\text{F-measure} = 2(recall_a \cdot precision)/(recall_a + precision)$$
$$recall_a = \#\text{ of corrected attributes} \;/\; \#\text{ of error attributes}$$
$$precision = \#\text{ of corrected attributes}/\#\text{ of changed attributes}$$

We compared the F-measure values of adopting $(Z, compT_c)$ with IncRep [CFG+07]. We remark that the *precision* of compCRegions is always 100%, if the user assures the correctness of attributes in certain regions, as defined.

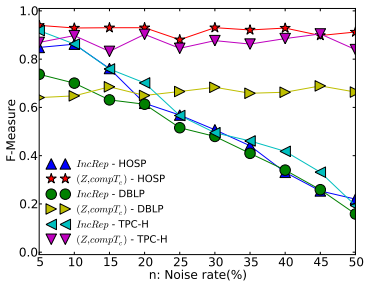
Figures 7.5(a), 7.5(b) and 7.5(c) show the results of F-measure comparisons when varying the parameters $d\%$, $|D_m|$ and $n\%$, respectively. Observe the following: (1)



(a) Varying $d\%$



(b) Varying $|D_m|$



(c) Varying $n\%$

Figure 7.5: F-Measure *w.r.t.* $d\%$, $|D_m|$ and $n\%$

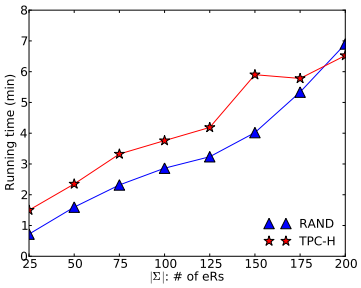
in most cases, when varying the three parameters described above, the F-measure of $(Z, compT_c)$ is better than that of IncRep, for all the datasets. This tells us that the certain regions and master data are more effective in guaranteeing the correctness of fixes than up-to-date techniques without leveraging master data, *e.g.,* IncRep. (2) Even when $|D_m|$ is small, $(Z, compT_c)$ can leverage D_m and perform reasonably well, if D_m can match a large part of certain regions of input tuples (*e.g., d%* is 80%), as depicted in Fig. 7.5(b). (3) The certain regions derived by compCRegions is insensitive to the noise rate, whereas IncRep is sensitive, as verified in Fig. 7.5(c).

This set of experiments verified that the proposed method performed well in fixing errors in data while assuring its correctness. The results also validate that the two most important factors are $d\%$ and D_m. When $|D_m|$ is large and $d\%$ is high, certain fixes could be expected.

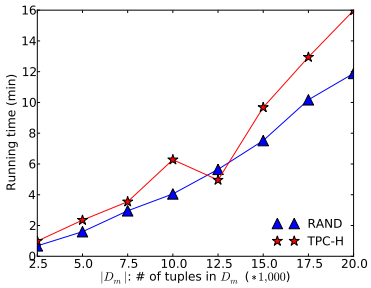
Exp-2: Efficiency and scalability. We evaluated the efficiency and scalability of both compCRegions in this set of experiments.

Since real-life data is not flexible to vary the three parameters $|D_m|$, $|\Sigma|$ and K (the number of maximal cliques), we used TPC-H and RAND data.

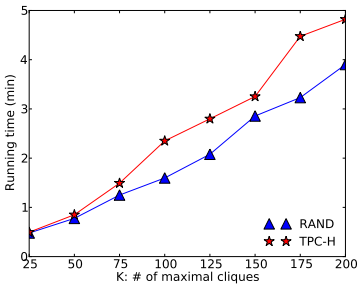
For TPC-H data, we have 55 eRs. When varying $|\Sigma|$, we randomly assigned these rules with pattern tuples so that we could always reach the number of eRs needed. For the RAND data, the default setting of $|R_m|$, $|R|$, $|D_m|$, $|\Sigma|$, $|\mathsf{LHS}|$ and K are 40, 20, 50, 5000, 4 and 100, respectively. When varying one parameter, the others were fixed.



(a) Running time *w.r.t.* $|\Sigma|$



(b) Running time *w.r.t.* $|D_m|$



(c) Running time *w.r.t.* K

Figure 7.6: The Scalability *w.r.t.* $|\Sigma|$, $|D_m|$ and K

We only report here the impact of the three most important factors: $|\Sigma|$, $|D_m|$ and K. The results for the other parameters are omitted due to space limitations.

Figures 7.6(a), 7.6(b), and 7.6(c) show the running time of computing compCRegions when varying $|\Sigma|$, $|D_m|$ and K, respectively. From these figures we can see the following.

(1) In all the cases, compCRegions could be computed efficiently. Note that for all datasets, Algorithm compCRegions was executed only once, irrelevant to the size of input data to be fixed. Therefore, it could be considered as a pre-computation. The time in minute level is thus acceptable.

(2) Figures 7.6(a) and 7.6(c) show sub-linear scalability, and better still, Figure 7.6(b) shows super-linear scalability. The trends in these results match our complexity analysis in Section 7.4.2, *i.e.,* in $O(|\Sigma|^2|D_m|\log|D_m| + K|\Sigma|^3 + K|\Sigma||D_m|\log|D_m|)$ time. This indicates that Algorithm compCRegions is scalable, and works well in practice.

Summary. From the experimental results we found the following. (1) The certain regions derived by our algorithm are effective and of high quality: at both the tuple level and the attribute level, our experiments have verified that the algorithm works well even with limited master data and high noise rate. (2) The completeness of master data (the amount of master data available) is critical to computing certain fixes. (3) Our algorithm scales well with the sizes of master data, editing rules and the number of certain regions.

APPENDIX: Additional Materials for the Experimental Study

We present more details on the datasets, the eRs that we designed for each data set, and the algorithm IncRep.

Datasets and editing rules.

(1) HOSP *data*. The data is maintained by the U.S. Department of Health & Human Services, and comes from hospitals that have agreed to submit quality information for Hospital Compare to make it public.

There are three tables: HOSP, HOSP_MSR_XWLK, and STATE_MSR_AVG, where (a) HOSP records the hospital information, including id (provider number, its ID), hName (hospital name), phn (phone number), ST (state), zip (ZIP code), and address; (b) HOSP_MSR_XWLK records the score of each measurement on each hospital in HOSP, *e.g.,* mName (measure name), mCode (measure code), and Score (the score of the measurement for this hospital); and (c) STATE_MSR_AVG records the average score of each measurement on hospitals in all US states, *e.g.,* ST (state), mName (measure name), sAvg (state average, the average score of all the hospitals in this state).

We created a big table by joining the three tables with *natural join*, among which we chose 19 attributes as the schema of both the master relation R_m and the relation R. We designed 37 eRs in total for the HOSP data, among which five important ones are listed 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 *data*. The DBLP service is well known for providing bibliographic information on major computer science journals and conferences. We first transformed the XML-formatted data into relational data. 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 as the schema of both the master relation

R_m and the 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), and publisher. We designed 16 eRs for the DBLP data, shown below.

φ_1 : ((a1, a1) $\rightarrow$ (hp1, hp1), t_{p1}[a1] = ($\overline{\text{nil}}$));
φ_2 : ((a2, a1) $\rightarrow$ (hp2, hp1), t_{p2}[a2] = ($\overline{\text{nil}}$));
φ_3 : ((a2, a2) $\rightarrow$ (hp2, hp2), t_{p3}[2] = ($\overline{\text{nil}}$));
φ_4 : ((a1, a2) $\rightarrow$ (hp1,hp2), t_{p4}[a2] = ($\overline{\text{nil}}$));
φ_5 : (((type, btitle, year), (type, btitle, year)) $\rightarrow$ (A, A), t_{p5}[type] = ('*conference*'));
φ_6 : (((type, crossref), (type, crossref) $\rightarrow$ (B, B), t_{p6}[type] = ('*conference*'));
φ_7 : (((type, a1, a2, title, pages), (type, a1, a2, title, pages)) $\rightarrow$
(C, C), t_{p7}[type] = ('*conference*')).

Here the attributes A, B and C range over the sets {isbn, publisher, crossref}, {btitle, year, isbn, publisher} and {isbn, publisher, year, btitle, 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.

(3) TPC-H *data.* The TPC Benchmark^{TM}H (TPC-H) is a benchmark for decision support systems. We created a big table by joining eight tables based on their foreign keys. The schema of both the master relation R_m and the relation R is the same as the one of the big table consisting of 58 attributes, *e.g.,* okey (order key), pkey (part key), num (line number), tprice (order total price), ckey (customer key), and skey (supplier key).

We designed 55 eRs all with empty pattern tuples. Since the data was the result of joining eight tables on foreign keys, we designed all eRs based on the foreign key attributes. We selectively report four eRs in the following.

φ_1 : (((okey, pkey), (okey, pkey)) $\rightarrow$ (num, num), t_{p1} = ());
φ_2 : ((okey, okey) $\rightarrow$ (tprice, tprice), t_{p2} = ());
φ_3 : ((ckey, ckey) $\rightarrow$ (name, name), t_{p3} = ());
φ_4 : ((skey, skey) $\rightarrow$ (address, address), t_{p4} = ()).

Adding noise. In the attribute level experiments, we added noises to the three datasets. The noise rate is defined as the ratio of (# of dirty attributes)/(# of total attributes). For each attribute that the noise was introduced, we kept the edit distance between the dirty value and the clean value less or equal than 3.

Algorithm IncRep. We implemented the incremental repairing algorithm IncRep in [CFG+07] to compare with the method proposed in this paper. Below we simply illustrate the algorithm IncRep (please see [CFG+07] for more details).

Taking as input a clean database D, a set ΔD of (possibly dirty) updates, a set Σ of CFDs, and an ordering O on ΔD, it works as follows. It first initializes the repair $Repr$ with the current clean database D. It then invokes a procedure called TupleResolve to repair each tuple t in ΔD according to the given order O, and adds the local repair $Repr_t$ of t to $Repr$ before moving to the next tuple. Once all the tuples in ΔD are processed, the final repair is returned as $Repr$. The key characteristics of IncRep are
(i) the repair grows at each step, providing in this way more information that can be used when cleaning the next tuple, and
(ii) the data in D is not modified since it is assumed to be clean.

For IncRep, we adopted the cost model presented in [CFG+07] based on the edit distance. For two values in the same domain, the cost model is defined as:

$$\mathsf{cost}\ (v, v') = w(t, A) \cdot \mathsf{dis}\ (v, v') / \mathsf{max}\ (|v|, |v'|),$$

where $w(t, A)$ is a *weight* in the range $[0, 1]$ associated with each attribute A of each tuple t in the dataset D.

For the cost of changing a tuple from t to t', we used the sum of $\mathsf{cost}\ (t[A], t[A'])$ for each A in the schema of R, *i.e.,* $\mathsf{cost}\ (t, t') = \Sigma_{A \in R}\ \mathsf{cost}\ (t[A], t'[A])$.

More specifically, in these experiments, we designed the CFDs based on the eRs that we have. Since the D_m and R have the same schemas in all datasets, we can easily design the corresponding CFDs from the eRs.

During the repair process, we enumerated one R tuple a time as ΔD. We then enlarged D to $Repr$, and repeated the process until all tuples were repaired. Because each time there was only one tuple in ΔD, we did not need to deal with the ordering O problem in IncRep.

Chapter 8

Conclusions and Future Work

In this chapter we summarize the results of this thesis, and propose future work.

8.1 Summary

The primary goal of this thesis has been to explore new classes of data dependencies for improving data quality. We conclude the results as below.

Problems	Dependencies	Infinite domain only
The satisfiability problem	INDs	$O(1)$ ([FV83])
	CINDs	$O(1)$ (Thm. 2.3.1)
The implication problem	INDs	PSPACE-complete ([AHV95, CFP84])
	CINDs	PSPACE-complete (Thm. 2.3.7)
	AINDs	NP-complete ([AHV95, CK84])
	ACINDs	NP-complete (Cor. 2.4.1)
	UINDs	PTIME ([AHV95, CKV90])
	UCINDs	PTIME (Thm. 2.5.1)
	acyclic UINDs	PTIME ([AHV95, CKV90])
	acyclic UCINDs	PTIME (Thm. 2.5.1)
The finite axiomatizability	INDs	IND1, IND2, IND3 ([AHV95, CFP84])
	CINDs	CIND1– CIND6 (Thm. 2.3.2)

Table 8.1: Summary of the results of CINDs in the absence of finite-domain attributes

Conditional inclusion dependencies (CINDs). We have proposed CINDs, a mild exten-

Problems	Dependencies	General setting
The satisfiability problem	INDs	$O(1)$ ([FV83])
	CINDs	$O(1)$ (Thm. 2.3.1)
The implication problem	INDs	PSPACE-complete ([AHV95, CFP84], Cor. 2.3.8)
	CINDs	EXPTIME-complete (Thm. 2.3.10)
	AINDs	NP-complete ([AHV95, CK84], Cor. 2.4.2)
	ACINDs	PSPACE-complete (Thm. 2.4.3)
	UINDs	PTIME ([AHV95, CKV90], Cor. 2.5.2)
	UCINDs	coNP-complete (Thm. 2.5.3)
	acyclic UINDs	PTIME ([AHV95, CKV90])
	acyclic UCINDs	coNP-complete (Cor. 2.5.4)
The finite axiomatizability	INDs	IND1, IND2, IND3 ([AHV95, CFP84])
	CINDs	CIND1– CIND8 (Thm. 2.3.4)

Table 8.2: Summary of the results of CINDs in the general setting

sion of INDs that is important in both contextual schema matching and data cleaning. We have also settled several fundamental problems associated with static analysis of CINDs, from the satisfiability to the finite axiomatizability, and to the implication problem.

Tables 8.1 and 8.2 summarize the main results for the analysis of CINDs established in this paper, compared with their counterparts for standard INDs. For the implication analysis, in particular, we have developed a sound and complete inference system for CINDs. We have also provided a complete picture of complexity bounds for the implication analysis of CINDs and INDs, focusing on the following dichotomies:

- with constant patterns (CINDs) vs. their absence (INDs),
- general CINDs vs. ACINDs and UCINDs; and
- the absence of finite-domain attributes vs. the general setting in which finite-domain attributes may be present.

We have investigated the impact of these factors on the implication analysis of inclusion dependencies. (a) For traditional INDs, AINDs, UINDs and acyclic UINDs, the presence of finite-domain attributes does not complicate the implication analysis. (b) The presence of constant patterns alone does not increase the complexity. Indeed, the implication problem for CINDs, ACINDs, UCINDs and acyclic UCINDs in the absence of finite-domain attributes retains the same complexity as its counterpart for INDs, AINDs,

Σ	General setting		Infinite domain only	
	Satisfiability	Implication	Satisfiability	Implication
CFDs [FGJK08]	NP-complete	coNP-complete	PTIME	PTIME
eCFDs	NP-complete (Thm. 3.2.1)	coNP-complete (Thm. 3.2.2)	NP-complete (Thm. 3.2.3)	coNP-complete (Thm. 3.2.3)
CFD^{c}s	NP-complete (Thm. 3.3.1)	coNP-complete (Thm. 3.3.1)	PTIME * (Thm. 3.3.4)	coNP-complete* (Thm. 3.3.5)
CFD^{p}s	NP-complete (Thm. 3.4.1)	coNP-complete (Thm. 3.4.5)	NP-complete (Thm. 3.4.2)	coNP-complete (Thm. 3.4.6)
CINDs [BFM07]	$O(1)$	EXPTIME-complete	$O(1)$	PSPACE-complete
CIND^{p}s	$O(1)$ (Prop. 3.4.3)	EXPTIME-complete (Thm. 3.4.7)	$O(1)$ (Prop. 3.4.3)	EXPTIME-complete (Thm. 3.4.8)
CFDs + CINDs [BFM07]	undecidable	undecidable	undecidable	undecidable
CFD^{p}s + CIND^{p}s	undecidable (Cor. 3.4.4)	undecidable (Cor. 3.4.9)	undecidable (Cor. 3.4.4)	undecidable (Cor. 3.4.9)

* We studied *bounded* CFD^{c}s here, where a set Σ of CFD^c is said to be *bounded by a constant k* if at most k attributes in the CFD^{c}s of Σ have a finite domain. In particular, when $k = 0$, all CFD^{c}s in Σ are defined in terms of attributes with an infinite domain.

Table 8.3: Summary of the complexity results of extensions of CFDs and CINDs

UINDs, and acyclic UINDs, respectively. Nevertheless, (c) the presence of both constant patterns and finite-domain attributes makes our lives harder. Indeed, the implication problem for CINDs, ACINDs, UCINDs and acyclic UCINDs in the general setting has a higher complexity than its counterpart for INDs, AINDs, UINDs and acyclic UINDs, respectively. These tell us that it is the interaction between constant patterns and finite-domain attributes that complicates the implication analysis.

Extensions of CFDs and CINDs. We have proposed (a) eCFDs, CFD^{c}s and CFD^{p}s, which are extensions of CFDs by supporting disjunctions and negations, cardinality constraints and synonym rules, and built-in predicates ($=, \neq, <, \leq, >$ and $\geq$), respectively, and (b) CIND^{p}s, which is an extension of CINDs by allowing patterns on data values to be expressed in terms of built-in predicates ($=, \neq, <, \leq, >$ and $\geq$). We have also studied two central problems associated with these dependencies: the satisfiability problem and the implication problem in the absence of finite-domain attributes or in the general setting.

The complexity bounds for reasoning about eCFDs, CFD^{c}s, CFD^{p}s and CIND^{p}s are summarized in Table 8.3. To give a complete picture we also include in Table 8.3 the

complexity bounds for the static analysis of CFDs and CINDs, taken from [FGJK08, BFM07]. The results shown in Table 8.4 tell us the following.

(a) Despite the increased expressive power, eCFDs, CFD^{c}s, CFD^{p}s and CIND^{p}s do not complicate the static analysis: the satisfiability and implication problems for eCFDs, CFD^{c}s and CFD^{p}s (resp. CIND^{p}s) have the same complexity bounds as their counterparts for CFDs (resp. CINDs), taken separately or together.

(b) In the special case when eCFDs, CFD^{c}s, CFD^{p}s and CIND^{p}s are defined with infinite-domain attributes only, however, the static analysis of eCFDs, CFD^{c}s and CFD^{p}s (resp. CIND^{p}s) do not get simpler, as opposed to their counterparts for CFDs (resp. CINDs). That is, in this special case the increased expressive power of eCFDs, CFD^{c}s, CFD^{p}s and CIND^{p}s comes at a price.

We have developed SQL-based methods to detect data inconsistencies based on eCFDs, CFD^{c}s, CFD^{p}s and CIND^{p}s, along with different methods to encode these data dependencies with data tables.

Dependency propagation for CFDs. This work is the first effort to study CFD propagation via views. The novelty of the work consists in the following: (a) a complete picture of complexity bounds on CFD propagation, for source dependencies given as either FDs or CFDs, and for views expressed in various fragments of relational algebra; (b) the first complexity results on dependency propagation when finite-domain attributes may be present, a practical and important problem overlooked by previous work; and (c) the first algorithm for computing minimal propagation covers for CFDs via SPC views in the absence of finite-domain attributes, without incurring more complexity than its counterparts for FDs and P views. Our experimental results have verified that the algorithm is efficient. These results are not only of theoretical interest, but also useful for data exchange, integration and cleaning.

We summarize in Table 8.4 complexity bounds on propagation from FDs to CFDs, and from CFDs to CFDs. To give a complete picture, we present the complexity bounds on propagation from FDs to FDs in Table 8.5, including the results from [Klu80, KP82, AHV95].

Matching dependencies (MDs). We have introduced a class of matching dependencies (MDs) and a notion of RCKs for record matching. As opposed to traditional dependencies, MDs and RCKs have a dynamic semantics and are defined in terms of similarity metrics, to accommodate errors and different representations in unreliable data sources.

Σ	View language	Complexity bounds	
		Infinite domain only	General setting
Propagation from FDs to CFDs			
FDs	SP	PTIME (Thm. 5.3.3)	PTIME (Thm. 5.3.5)
	SC	PTIME (Thm. 5.3.3)	coNP-complete (Thm. 5.3.5)
	PC	PTIME (Thm. 5.3.3)	PTIME (Thm. 5.3.5)
	SPC	PTIME (Thm. 5.3.3)	coNP-complete (Thm. 5.3.5)
	SPCU	PTIME (Thm. 5.3.3)	coNP-complete (Thm. 5.3.5)
	RA	undecidable (Thm. 5.3.3)	undecidable (Thm. 5.3.5)
Propagation from CFDs to CFDs			
CFDs	S	PTIME (Thm. 5.3.7)	coNP-complete (Cor. 5.3.8)
	P	PTIME (Thm. 5.3.7)	coNP-complete (Cor. 5.3.8)
	C	PTIME (Thm. 5.3.7)	coNP-complete (Cor. 5.3.8)
	SPC	PTIME (Thm. 5.3.7)	coNP-complete (Cor. 5.3.8)
	SPCU	PTIME (Thm. 5.3.7)	coNP-complete (Cor. 5.3.8)
	RA	undecidable (Thm. 5.3.7)	undecidable (Cor. 5.3.8)

Table 8.4: Summary of the complexity results of CFD propagation

Propagation from FDs to FDs		
View language	Complexity bounds	
	Infinite domain only	General setting
SP	PTIME ([KP82, AHV95])	PTIME (Cor. 5.3.6)
SC	PTIME ([KP82, AHV95])	coNP-complete (Thm. 5.3.4)
PC	PTIME ([KP82, AHV95])	PTIME (Cor. 5.3.6)
SPCU	PTIME ([KP82, AHV95])	coNP-complete (Cor. 5.3.6)
RA	undecidable ([Klu80])	undecidable ([Klu80])

Table 8.5: Summary of the complexity results of FD propagation

To reason about MDs, we have proposed a deduction mechanism to capture their dynamic semantics, a departure from the traditional notion of implication. We have also provided a sound and complete inference system and efficient algorithms for deducing MDs and high quality RCKs, for matching, blocking and windowing. Our conclusion is that the techniques are a promising tool for improving match quality and efficiency, as verified by our experimental study.

Editing Rules (eRs). We have proposed editing rules that, in contrast to integrity constraints used in data cleaning, are able to find certain fixes by updating input tuples with master data. We have identified fundamental problems for deciding certain fixes and certain regions, and established their complexity bounds. We have also developed

a graph-based algorithm for deriving certain regions from editing rules and master data. As verified by our experimental study, these yield a promising method for fixing data errors while ensuring its correctness.

We are extending Quaid [CFG+07], our working system for data cleaning, to support master data and to experiment with real-life data that Quaid processes. We are also exploring optimization methods to improve our derivation algorithm. Another topic is to develop methods for discovering editing rules from sample inputs and master data, along the same lines as discovering other data quality rules [CM08, GKK+08].

8.2 Future work

There is naturally much more to be done. We list future research directions, and identify problems to be studied.

Interactions between CFDs and CINDs. It has been shown that when CINDs and CFDs are taken together, both the satisfiability problem and the implication problem become *undecidable* [BFM07]. Nevertheless, it is not known yet whether the problems are decidable when putting together CFDs (or FDs) and fragments of CINDs, *e.g.,* AINDs, ACINDs, UINDs, UCINDs, acyclic UINDs and acyclic UCINDs. We want to find out the impact of built-in predicates (*e.g.,* $=, \neq, <, \leq, >$ and $\geq$) on the static analysis of ACINDs and UCINDs.

Unifying extensions of CFDs. One topic for future work is to develop a dependency language that is capable of expressing various extensions of CFDs (*e.g.,* eCFDs, CFD^{c}s, and CFD^{p}s) without increasing the complexity of static analysis.

Data repairing based on both extensions of CFDs and CINDs. Both CFDs and CINDs, as well as their extensions, are useful in data cleaning. We have shown how to detect data inconsistencies with the proposed data dependencies, *i.e.,* CINDs, eCFDs, CFD^{c}s, CFD^{p}s and CIND^{p}s. And data repairing methods based on CFDs have been developed in [FGJK08]. However, effective and efficient data repairing is yet to be developed when both extensions of CFDs and CINDs are brought into the play.

Dependency discovery. It is important to develop effective algorithms for discovering CINDs and these extensions of CFDs and CINDs, along the same lines as their counterparts for CFDs [CM08, FGLX09, GKK+08].

Extensions of MDs. First, an extension of MDs is to support "negation", to specify when records *cannot* be matched. Second, one can augment similarity relations with constants, to capture domain-specific synonym rules along the same lines as [ACG02, ACK08]. Third, we have so far focused on 1-1 correspondences between attributes, as commonly assumed for record matching after data standardization [EIV07]. As observed in [DLD$^+$04], complex matches may involve correspondences between multiple attributes of one schema and one or more attributes of another. We are extending MDs to deal with such structural heterogeneity. Fourth, we are investigating, experimentally and analytically, the impact of different similarity metrics on match quality, and the impact of various quality models on deducing RCKs. Finally, an important and practical topic is to develop algorithms for discovering MDs from sample data, along the same lines as discovery of FDs. As remarked earlier, probabilistic methods such as EM algorithms [Jar89, Win02] suggests an effective approach to discovering MDs.

Unifying data repairing and record matching. Repairing aims to make a database consistent, *i.e.,* to satisfy a given set of integrity constraints, by incurring minimal updates to it. Matching is to identify tuples from unreliable data sources that refer to the same real-world object. Current data quality systems have endeavored to provide the functionality of repairing and matching, but treat them as separate processes. One future work is to seamlessly unify repairing and matching, based on integrity constraints (*e.g.,* CFDs and CINDs) and matching rules (*e.g.,* MDs).

Graph based complex object identification. The object identification problem is to identify tuples in one or more relations that refer to the same real-world entity. Using graphs to represent complex objects, such as Web sites and semi-structured data, this problem can be treated as the *graph matching* problem [FLM$^+$10a, FLM$^+$10c]. We explore new models to capture the matching semantics, and develop effective and efficient algorithms to find matching complex objects (graphs).

Bibliography

[AB96] Tatsuya Akutsu and Feng Bao. Approximating minimum keys and optimal substructure screens. In *Proceedings of the Annual International Conferenceon on Computing and Combinatorics (COCOON)*, 1996.

[ABC+03a] M. Arenas, L. E. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *Theoretical Computer Science (TCS)*, 296(3):405–434, 2003.

[ABC03b] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. *Theory and Practice of Logic Programming (TPLP)*, 3(4-5):393–424, 2003.

[ACG02] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.

[ACK08] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Transformation-based framework for record matching. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2008.

[ADS09] Jennifer Widom Anish Das Sarma, Jeffrey Ullman. Schema design for uncertain databases. In *Proceedings of the 3rd Alberto Mendelzon Workshop on Foundations of Data Management*, 2009.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[APRR09] Marcelo Arenas, Jorge Pérez, Juan Reutter, and Cristian Riveros. Composition and inversion of schema mappings. *SIGMOD Record*, 38(3), 2009.

[ARS09] Arvind Arasu, Christopher Re, and Dan Suciu. Large-scale deduplication with constraints using Dedupalog. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2009.

[ASU79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing (SICOMP)*, 8(2):218–246, 1979.

[BB79] Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems (TODS)*, 4(1):30–59, 1979.

[BBFL08] L Bertossi, L Bravo, E. Franconi, and A Lopatenko. The complexity and approximation of fixing numerical attributes in databases under integrity constraints. *Information Systems*, 33(4):407–434, 2008.

[BCW99] Marianne Baudinet, Jan Chomicki, and Pierre Wolper. Constraint-Generating Dependencies. *Journal of Computer and System Sciences (JCSS)*, 59(1):94–115, 1999.

[BEFF06] Philip Bohannon, Eiman Elnahrawy, Wenfei Fan, and Michael Flaster. Putting context into schema matching. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2006.

[Ber06] Leopoldo Bertossi. Consistent query answering in databases. *SIGMOD Record*, 35(2):68–76, 2006.

[BFFR05] Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[BFGM08] Loreto Bravo, Wenfei Fan, Floris Geerts, and Shuai Ma. Increasing the expressivity of conditional functional dependencies without extra complexity. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2008.

[BFM07] Loreto Bravo, Wenfei Fan, and Shuai Ma. Extending dependencies with conditions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[BP83] Paul De Bra and Jan Paredaens. Conditional dependencies for horizontal decompositions. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 1983.

[BS06] Carlo Batini and Monica Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[BV84] Catriel Beeri and Moshe Vardi. A proof procedure for data dependencies. *Journal of the ACM (JACM)*, 31(4):718–741, 1984.

[BV06] Radim Belohlávek and Vilém Vychodil. Data tables with similarity relations: Functional dependencies, complete rules and non-redundant bases. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2006.

[CAM07] Bogdan Cautis, Serge Abiteboul, and Tova Milo. Reasoning about XML update constraints. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2007.

[CCGK07] Surajit Chaudhuri, Bee-Chung Chen, Venkatesh Ganti, and Raghav Kaushik. Example-driven design of efficient record matching queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[CFG+07] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[CFJM06] Gao Cong, Wenfei Fan, Xibei Jia, and Shuai Ma. Prata: A system for xml publishing, integration and view maintenance (poster paper). In *Proceedings of the UK e-Science All Hands Meeting, Nottingham*, 2006.

[CFM09a] Wenguang Chen, Wenfei Fan, and Shuai Ma. Analyses and validation of conditional dependencies with built-in predicates. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, 2009.

[CFM09b] Wenguang Chen, Wenfei Fan, and Shuai Ma. Incorporating cardinality constraints and synonym rules into conditional functional dependencies. *Information Processing Letters (IPL)*, 109(14):783–789, 2009.

[CFP84] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *Journal of Computer and System Sciences (JCSS)*, 28(1):29–59, 1984.

[CGGM03] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2003.

[Chl86] Bogdan S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences (JCSS)*, 32(3):374–392, 1986.

[Cho07] Jan Chomicki. Consistent query answering: Five easy pieces. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2007.

[CK84] Stavros S. Cosmadakis and Paris C. Kanellakis. Functional and inclusion dependencies a graph theoretic approach. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 1984.

[CKV90] Stavros S. Cosmadakis, Paris C. Kanellakis, and Moshe Y. Vardi. Polynomial-time implication problems for unary inclusion dependencies. *Journal of the ACM (JACM)*, 37(1):15–46, 1990.

[CL94] D. Calvanese and M. Lenzerini. On the interaction between isa and cardinality constraints. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 1994.

[CM05] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, 2005.

[CM08] Fei Chiang and Renée J. Miller. Discovering data quality rules. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[CR02] William W. Cohen and Jacob Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2002.

[CSGK07] Surajit Chaudhuri, Anish Das Sarma, Venkatesh Ganti, and Raghav Kaushik. Leveraging aggregate constraints for deduplication. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2007.

[DFHQ03] Susan Davidson, Wenfei Fan, Carmem Hara, and Jing Qin. Propagating XML constraints to relations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2003.

[DLD+04] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Y. Halevy, and Pedro Domingos. iMAP: Discovering complex mappings between database schemas. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[DPT06] Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[Eck02] Wayne W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. The Data Warehousing Institute, 2002.

[EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Data and Knowledge Engineering (TKDE)*, 19(1):1–16, 2007.

[Fag82] Ronald Fagin. Horn clauses and database dependencies. *Journal of the ACM (JACM)*, 29(4):952–985, 1982.

[Fan08] Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2008.

[FGJ+] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. Dynamic constraints for record matching. *The VLDB Journal*. To appear.

[FGJK08] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2), 2008.

[FGLX] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE Transactions on Data and Knowledge Engineering (TKDE)*. To appear.

[FGLX09] Wenfei Fan, Floris Geerts, Laks V.S. Lakshmanan, and Ming Xiong. Discovering conditional functional dependencies. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2009.

[FGMM10] Wenfei Fan, Floris Geerts, Shuai Ma, and Heiko Müller. Detecting inconsistencies in distributed data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2010.

[FH76] Ivan Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *Journal of the American Statistical Association (JASA)*, 71(353):17–35, 1976.

[FJLM09] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1), 2009.

[FJT83] Patrick C. Fischer, Jiann H. Jou, and Don-Min Tsou. Succinctness in dependency systems. *Theoretical Computer Science (TCS)*, 24:323–329, 1983.

[FLM+10a] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[FLM+10b] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[FLM+10c] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[FLM+11] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2011.

[FM] F-Measure. *http://en.wikipedia.org/wiki/F-measure*.

[FMH+08] Wenfei Fan, Shuai Ma, Yanli Hu, Jie Liu, and Yinghui Wu. Propagating functional dependencies with conditions. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[FPL+01] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR)*, 2001.

[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 *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2010.

[Fra] Fraud. http://www.sas.com/industry/fsi/fraud/.

[FS69] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the American Statistical Association (JASA)*, 64(328):1183–1210, 1969.

[FV83] Ronald Fagin and Moshe Y. Vardi. Armstrong databases for functional and inclusion dependencies. *Information Processing Letters (IPL)*, 16(1):13–19, 1983.

[FV84] Ronald Fagin and Moshe Y. Vardi. The theory of data dependencies - an overview. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 1984.

[Gar07] Gartner. Forecast: Data quality tools, worldwide, 2006-2011. Technical report, Gartner, 2007.

[GFS+01] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model

and algorithms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[Gil88] P Giles. A model for generalized edit and imputation of survey data. *The Canadian Journal 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 J. Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[GKMS04] Sudipto Guha, Nick Koudas, Amit Marathe, and Divesh Srivastava. Merging the results of approximate match operations. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[GM85] John Grant and Jack Minker. Normalization and axiomatization for numerical dependencies. *Information and Control*, 65(1):1–17, 1985.

[Got87] Georg Gottlob. Computing covers for embedded functional dependencies. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 1987.

[Gra91] Gösta Grahne. *The Problem of Incomplete Information in Relational Databases*. Springer, 1991.

[GS85] Seymour Ginsburg and Edwin H. Spanier. On completing tables to satisfy functional dependencies. *Theoretical Computer Science (TCS)*, 39:309–317, 1985.

[GZ82] Seymour Ginsburg and Sami Zaiddan. Properties of functional-dependency families. *Journal of the ACM (JACM)*, 29(3):678–698, 1982.

[HHH+05] Laura Haas, Mauricio Hernández, Howard Ho, Lucian Popa, and Mary Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[HL04] Qi He and TokWang Ling. Extending and inferring functional dependencies in schema transformation. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*, 2004.

[HS95] Mauricio A. Hernndez and Salvatore J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1995.

[HSW09] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. *Data Quality and Record Linkage Techniques*. Springer, 2009.

[Hul85] Richard Hull. Non-finite specifiability of projections of functional dependency families. *Theoretical Computer Science (TCS)*, 39:239–265, 1985.

[IL84] Tomasz Imieliński and Witold Lipski Jr. Incomplete information in relational databases. *Journal of the ACM (JACM)*, 31(4):761–791, 1984.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing (SICOMP)*, 17(5):935–938, 1988.

[Jar89] M.A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa florida. *Journal of the American Statistical Association (JASA)*, 89:414–420, 1989.

[JK84] David S. Johnson and Anthony C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences (JCSS)*, 28(1):167–189, 1984.

[JPY88] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Information Processing Letters (IPL)*, 27(3):119–123, 1988.

[Kan80] P. C. Kanellakis. On the computational complexity of cardinality constraints in relational databases. *Information Processing Letters (IPL)*, 11(2):98–101, 1980.

[KL09] Solmaz Kolahi and Laks Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2009.

[Klu80] Anthony C. Klug. Calculating constraints on relational expressions. *ACM Transactions on Database Systems (TODS)*, 5(3):260–290, 1980.

[Kol05a] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2005.

[Kol05b] Phokion G. Kolaitis. Schema mappings, data exchange, and metadata management. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2005.

[KP82] Anthony C. Klug and Rod Price. Determining view dependencies using tableaux. *ACM Transactions on Database Systems (TODS)*, 7(3):361–380, 1982.

[KSSV09] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. Metric functional dependencies. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2009.

[LB07] Andrei Lopatenko and Leopoldo Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2007.

[Len02] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 2002.

[LO78] Claudio L Lucchesi and Sylvia L Osborn. Candidate keys for relations. *Journal of Computer and System Sciences (JCSS)*, 17(2):270–279, 1978.

[Los09] David Loshin. *Master Data Management*. Knowledge Integrity, Inc., 2009.

[LSPR96] Ee-Peng Lim, Jaideep Srivastava, Satya Prabhakar, and James Richardson. Entity identification in database integration. *Information Sciences*, 89(1-2):1–38, 1996.

[Mah97] Michael J. Maher. Constrained dependencies. *Theoretical Computer Science (TCS)*, 173(1):113–149, 1997.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[MS96] Michael J. Maher and Divesh Srivastava. Chasing Constrained Tuple-Generating Dependencies. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 1996.

[MU04] Kazuhisa Makino and Takeaki Uno. New algorithms for enumerating all maximal cliques. In *Proceedings of the candinavian Workshop on Algorithm Theory (SWAT)*, 2004.

[Pap94] Christos H Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[PT99] Lucian Popa and Val Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1999.

[Ram98] Raghu Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 1998.

[RD00] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[Red98] Thomas Redman. The impact of poor data quality on the typical enterprise. *Communications of the ACM (CACM)*, 2:79–82, 1998.

[RW08] J. Radcliffe and A. White. Key issues for master data management. Technical report, Gartner, 2008.

[Sav70] Walter J. Savitch. Relationships between Nondeterministic and Deterministic Tape Complexities. *Journal of Computer and System Sciences (JCSS)*, 4:177–192, 1970.

[SB02] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2002.

[Sci86] Edward Sciore. Comparing the universal instance and relational data models. *Advances in Computing Research*, 3:139–162, 1986.

[SD05] Parag Singla and Pedro Domingos. Object identification with attribute-mediated dependences. In *Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)*, 2005.

[SK86] Abraham Silberschatz and Henry F. Korth. *Database System Concepts*. McGraw-Hill, 1986.

[SLD05] Warren Shen, Xin Li, and AnHai Doan. Constraint-based entity matching. In *Proceedings National Conference on Artificial Intelligence (AAAI)*, 2005.

[SMO07] G Sauter, B Mathews, and E Ostic. Information service patterns, part 3: Data cleansing pattern. IBM, 2007.

[Sou] Soundex. http://en.wikipedia.org/wiki/Soundex/.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4):633–655, 1980.

[Sze87] Róbert Szelepcsényi. The meothod of focing for nondeterministic automata. *Bulletin of the EATCS*, 33:96–99, 1987.

[TW05] David Toman and Grant E. Weddell. On reasoning about structural equality in XML: a description logic approach. *Theoretical Computer Science (TCS)*, 336(1):181–203, 2005.

[TW06] David Toman and Grant E. Weddell. On keys and functional dependencies as first-class citizens in description logics. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR)*, 2006.

[Ull82] Jeffrey D. Ullman. *Principles of Database Systems*. Computer Science Press, 1982.

[Val79] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing (SICOMP)*, 8(3):410–421, 1979.

[Vaz03] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[vEB97] Peter van Emde Boas. The convenience of tilings. In *Complexity, Logic, and Recursion Theory*. Marcel Dekker Inc, 1997.

[VEH02] Vassilios S. Verykios, Ahmed K. Elmagarmid, and Elias Houstis. Automating the approximate record-matching process. *Information Sciences*, 126(1-4):83–89, 2002.

[Via87] Victor Vianu. Dynamic functional dependencies and database aging. *Journal of the ACM (JACM)*, 34(1):28–59, 1987.

[WC96] Jennifer Widom and Stefano Ceri. *Active database systems: triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[Wij05] Jef Wijsen. Database repairing using updates. *ACM Transactions on Database Systems (TODS)*, 30(3):722–768, 2005.

[Win02] William E. Winkler. Methods for record linkage and bayesian networks. Technical Report RRS2002/05, U.S. Census Bureau, 2002.

[Win04] William E. Winkler. Methods for evaluating and creating data quality. *Information Systems*, 29(7):531–550, 2004.

[WNJ+08] Melanie Weis, Felix Naumann, Ulrich Jehle, Jens Lufter, and Holger Schuster. Industry-scale duplicate detection. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2), 2008.

[Yan07] William Yancey. BigMatch: A program for extracting probable matches from a large file. Technical Report Computing 2007/01, U.S. Census Bureau, 2007.