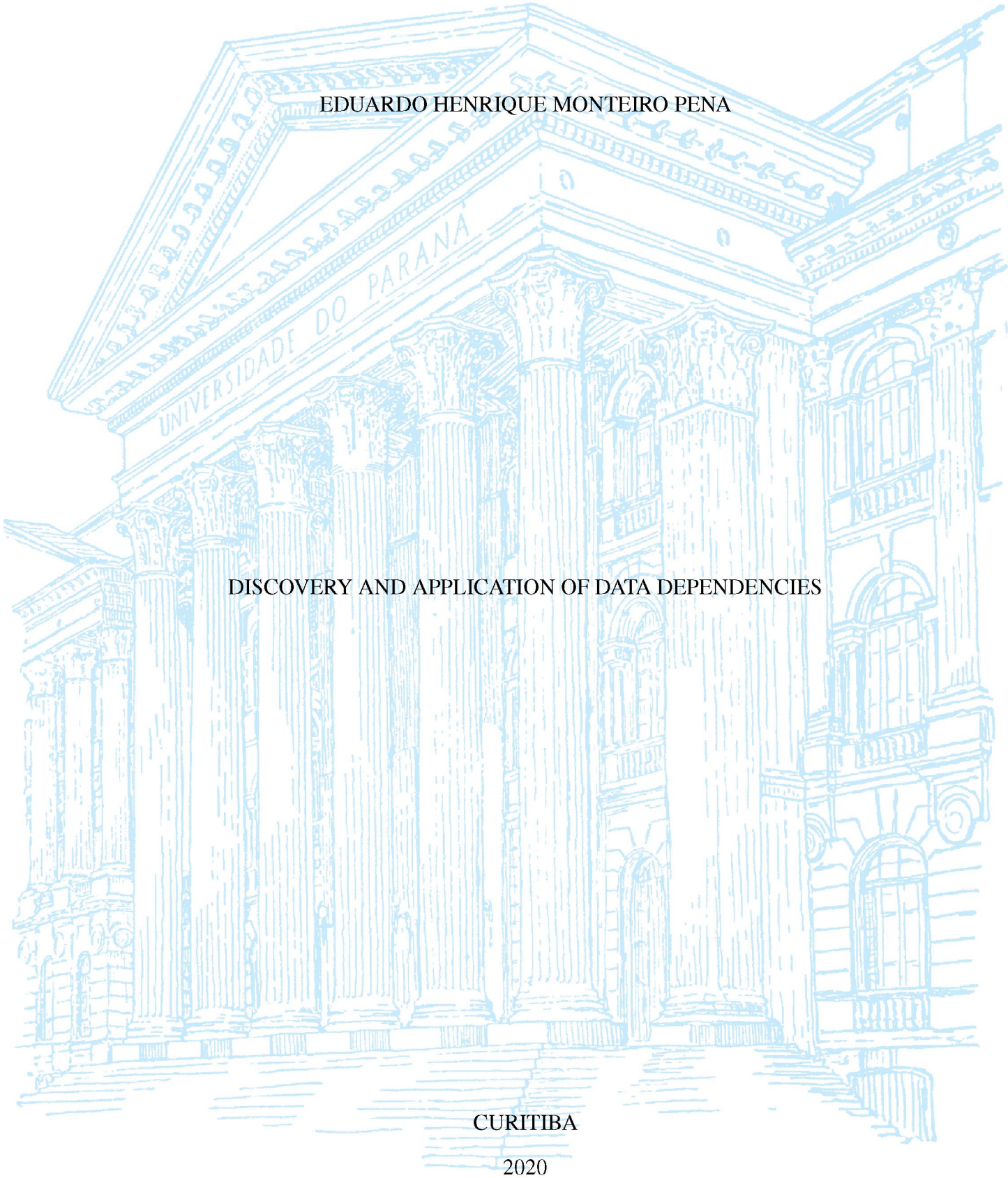
UNIVERSIDADE FEDERAL DO PARANÁ

EDUARDO HENRIQUE MONTEIRO PENA



DISCOVERY AND APPLICATION OF DATA DEPENDENCIES

CURITIBA

2020

EDUARDO HENRIQUE MONTEIRO PENA

DISCOVERY AND APPLICATION OF DATA DEPENDENCIES

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação.*

Orientador: Prof. Dr. Eduardo Cunha de Almeida.

CURITIBA

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

P397d Pena, Eduardo Henrique Monteiro
Discovery and application of data dependencies [recurso eletrônico] / Eduardo Henrique Monteiro Pena – Curitiba, 2020.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.

Orientador: Prof. Dr. Eduardo Cunha de Almeida

1. Banco de dados. 2. Qualidade de dados. 3. Perfilamento de dados. I. Universidade Federal do Paraná. II. Almeida, Eduardo Cunha de. III. Título.

CDD: 005.74

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **EDUARDO HENRIQUE MONTEIRO PENA** intitulada: **Discovery and Application of Data Dependencies**, sob orientação do Prof. Dr. EDUARDO CUNHA DE ALMEIDA, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 08 de Setembro de 2020.

Assinatura Eletrônica
09/09/2020 09:35:38.0
EDUARDO CUNHA DE ALMEIDA
Presidente da Banca Examinadora

Assinatura Eletrônica
09/09/2020 09:32:29.0
ALTIGRAN SOARES DA SILVA
Avaliador Externo (UNIVERSIDADE FEDERAL DO AMAZONAS)

Assinatura Eletrônica
09/09/2020 09:55:04.0
WAGNER HUGO BONAT
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
10/09/2020 10:31:14.0
HELENA GALHARDAS
Avaliador Externo (UNIVERSIDADE DE LISBOA)

Assinatura Eletrônica
15/09/2020 22:01:00.0
JULIANA FREIRE
Avaliador Externo (NEW YORK UNIVERSITY)

To my family

ACKNOWLEDGEMENTS

This thesis would have been impossible if it had not been for all the kind people I met along with its development. First, I would like to express my deepest appreciation to my advisor, Professor Eduardo Cunha de Almeida, who introduced me to the field of database systems research in such an inspirational form. I am especially grateful for Eduardo's support, patience, and guidance while allowing me freedom during my graduate experience. I hope to advise any future student who chooses to work with me similarly.

I also want to express my deepest gratitude to Professor Felix Naumann, who was my advisor during my internship at the Hasso Plattner Institut (HPI) in Potsdam-Germany. For someone who must deal with the many stresses of a major research institution daily, I cannot imagine someone being more selfless and caring more for his student' interests. I am deeply inspired by his interpersonal intelligence, wisdom, and dedication to his work.

I am deeply grateful to my thesis committee members: Professor Altigran Soares da Silva, Professor Helena Galhardas, Professor Juliana Freire, and Professor Wagner Hugo Bonat, for taking the time to read and to offer valuable suggestions to improve this thesis.

I would also like to thank all colleagues of the Database Lab at the Federal University of Paraná, all the colleagues of the Information System Group at HPI, and all the co-workers at the Federal University of Technology - Paraná. I feel fortunate to have overlapped with such wonderful people during my doctoral time.

I am deeply grateful for the support and encouragement from my friends from Brazil, and all the international friends I made in Germany.

Finally, this thesis would not have been possible without the support and love of my family. You have always been there for me. To you all, I dedicate this work.

RESUMO

Dependências de dados (ou, simplesmente, dependências) têm um papel fundamental em muitos aspectos do gerenciamento de dados. Em consequência, pesquisas recentes têm desenvolvido contribuições para importante problemas relacionados à dependências. Esta tese traz contribuições que abrangem dois desses problemas.

O primeiro problema diz respeito à descoberta de dependências com alto poder de expressividade. O objetivo é substituir o projeto manual de dependências, o qual é sujeito a erros, por um algoritmo capaz de descobrir dependências a partir de dados apenas. Nesta tese, estudamos a descoberta de restrições de negação, um tipo de dependência que contorna muitos problemas relacionados ao poder de expressividade de depêndencias. As restrições de negação têm poder de expressividade suficiente para generalizar outros tipos importantes de dependências, e expressar complexas regras de negócios. No entanto, sua descoberta é computacionalmente difícil, pois possui um espaço de busca maior do que o espaço de busca visto na descoberta de dependências mais simples. Esta tese apresenta novas técnicas na forma de um algoritmo para a descoberta de restrições de negação. Avaliamos o projeto de nosso algoritmo em uma variedade de cenários: conjuntos de dados reais e sintéticos; e números variáveis de registros e colunas. Nossa avaliação mostra que, em comparação com soluções do estado da arte, nosso algoritmo melhora significativamente a eficiência da descoberta de restrição de negação em termos de tempo de execução.

O segundo problema diz respeito à aplicação de dependências no gerenciamento de dados. Primeiro, estudamos a aplicação de dependências na melhoraria da consistência de dados, um aspecto crítico da qualidade dos dados. Uma maneira comum de modelar inconsistências é identificando violações de dependências. Nesse contexto, esta tese apresenta um método que estende nosso algoritmo para a descoberta de restrições de negação de forma que ele possa retornar resultados confiáveis, mesmo que o algoritmo execute sobre dados contendo alguns registros inconsistentes. Mostramos que é possível extrair evidências dos conjuntos de dados para descobrir restrições de negação que se mantêm aproximadamente. Nossa avaliação mostra que nosso método retorna dependências de negação que podem identificar, com boa precisão e recuperação, inconsistências no conjunto de dados de entrada.

Esta tese traz mais uma contribuição no que diz respeito à aplicação de dependências para melhorar a consistência de dados. Ela apresenta um sistema para detectar violações de dependências de forma eficiente. Realizamos uma extensa avaliação de nosso sistema usando

comparações com várias abordagens; dados do mundo real e sintéticos; e vários tipos de restrições de negação. Mostramos que os sistemas de gerenciamento de banco de dados comerciais testados começam a apresentar baixo desempenho para conjuntos de dados relativamente pequenos e alguns tipos de restrições de negação. Nosso sistema, por sua vez, apresenta execuções até três ordens de magnitude mais rápidas do que as de outras soluções relacionadas, especialmente para conjuntos de dados maiores e um grande número de violações identificadas.

Nossa contribuição final diz respeito à aplicação de dependências na otimização de consultas. Em particular, esta tese apresenta um sistema para a descoberta automática e seleção de dependências funcionais que potencialmente melhoram a execução de consultas. Nosso sistema combina representações das dependências funcionais descobertas em um conjunto de dados com representações extraídas de cargas de trabalho de consulta. Essa combinação direciona a seleção de dependências funcionais que podem produzir reescritas de consulta para as consultas de entrada. Nossa avaliação experimental mostra que nosso sistema seleciona dependências funcionais relevantes que podem ajudar na redução do tempo de resposta geral de consultas.

Palavras-chave: Perfilamento de dados. Qualidade de dados. Limpeza de dados. Depenência de dados. Execução de consulta.

ABSTRACT

Data dependencies (or dependencies, for short) have a fundamental role in many facets of data management. As a result, recent research has been continually driving contributions to central problems in connection with dependencies. This thesis makes contributions that reach two of these problems.

The first problem regards the discovery of dependencies of high expressive power. The goal is to replace the error-prone process of manual design of dependencies with an algorithm capable of discovering dependencies using only data. In this thesis, we study the discovery of denial constraints, a type of dependency that circumvents many expressiveness drawbacks. Denial constraints have enough expressive power to generalize other important types of dependencies and to express complex business rules. However, their discovery is computationally hard since it regards a search space that is bigger than the search space seen in the discovery of simpler dependencies. This thesis introduces novel algorithmic techniques in the form of an algorithm for the discovery of denial constraints. We evaluate the design of our algorithm in a variety of scenarios: real and synthetic datasets; and a varying number of records and columns. Our evaluation shows that, compared to state-of-the-art solutions, our algorithm significantly improves the efficiency of denial constraint discovery in terms of runtime.

The second problem concerns the application of dependencies in data management. We first study the application of dependencies for improving data consistency, a critical aspect of data quality. A common way to model data inconsistencies is by identifying violations of dependencies. In that context, this thesis presents a method that extends our algorithm for the discovery of denial constraints such that it can return reliable results even if the algorithm runs on data containing some inconsistent records. A central insight is that it is possible to extract evidence from datasets to discover denial constraints that almost hold in the dataset. Our evaluation shows that our method returns denial dependencies that can identify, with good precision and recall, inconsistencies in the input dataset.

This thesis makes one more contribution regarding the application of dependencies for improving data consistency. It presents a system for detecting violations of dependencies efficiently. We perform an extensive evaluation of our system that includes comparisons with several different approaches; real-world and synthetic data; and various kinds of denial constraints. We show that the tested commercial database management systems start underperforming for relatively small datasets and production dependencies in the form of denial constraints. Our

system, in turn, is up to three orders-of-magnitude faster than related solutions, especially for larger datasets and massive numbers of identified violations.

Our final contribution regards the application of dependencies in query optimization. In particular, this thesis presents a system for the automatic discovery and selection of functional dependencies that potentially improve query executions. Our system combines representations from the functional dependencies discovered in a dataset with representations of the query workloads that run for that dataset. This combination guides the selection of functional dependencies that can produce query rewritings for the incoming queries. Our experimental evaluation shows that our system selects relevant functional dependencies, which can help in reducing the overall query response time.

Keywords: Data profiling. Data quality. Data cleaning. Data dependencies. Query execution.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

Chapter 1

Dependencies in databases

Database management systems have become ubiquitous in computer systems, from personal computers to enterprise computing platforms. As a consequence, they have been evolving to meet the requirements of a variety of applications from various segments. For example, modern applications often require a fast response to queries and assume that database management systems can guarantee a certain degree of reliability or quality for their query answers. This evolution has been fostering the development of a large body of concepts and techniques in data management in general.

One of the many essential aspects of relational database management systems regards their capability of enforcement of *constraints* on database objects. Constraints represent knowledge about the application domain and define restrictions on the actual values of database instances. An important category of constraints is *data dependencies* (or *dependencies* for short), because they describe the semantics of databases. Dependencies are necessary because the relational model, by itself, lacks artifacts that guide the semantic interpretation of tables. The tuples in a table represent collections of related data values, which, in turn, represent facts on entities or relationships in the real world. Although the names of tables and columns can help us to grasp preliminary meanings of the values in each tuple, they do not specify how these values are related to each other or how we would characterize invalid values. Dependencies can incorporate such semantics into the relational model. In turn, relational database management systems can enforce some types of dependencies as constraints to restrict data inconsistencies and enhance *data quality*. In the following, we outline a few fundamentals about dependencies that give context for the main contributions of this thesis.

A critical dimension in data quality is *data consistency*. Fan [1] gives a concise definition: "Data consistency refers to the validity and integrity of data representing real-world entities". By restricting inconsistencies, database management systems guarantee access to higher quality data, which is essential in supporting reliable query answers.

Preventing the storage of invalid or inconsistent data is a battle on many fronts, for instance, users with a lack of application knowledge, machine-to-machine data inputs with errors, data evolution, or data integration scenarios, to name but a few. This battle attracts broad interest.

The development of mechanisms to improve the integrity of data has long been a vivid topic in both academic research and industry-based projects [2, 3, 4, 5].

Commercial database management systems support a few types of dependencies known as *integrity constraints*. Once the database designers or users have a database project ready, they can create and maintain integrity constraints using the structured query language (SQL). The database management system then needs to maintain data integrity by restricting those database updates that do not adhere to the database's integrity constraints. Most database management systems implement only traditional integrity constraints: domain constraints, key constraints, and foreign key constraints. Unfortunately, these types of dependencies cannot identify many critical data inconsistencies, as we show with the following examples.

Consider the salesReps schema and the sales reps tuples in Table 1.1; and assume column ID as the primary key, and column SID as a foreign key referencing salesReps on ID. Also, assume that there are no issues with the domain in column values. The data conform to traditional integrity constraints. However, a database designer with experience in the application domain would still spot critical inconsistencies. For example, if any two tuples have the same value combination in address, city, and state (ST), then they should have the same value in zip code (Zip). Tuples t_1 and t_2 are inconsistent with this statement. Also, zip code uniquely determines state (and city), thus, tuples t_1 and t_3 are inconsistent. As another example, assume that sales reps cannot earn higher salaries than their supervisors—again, tuples t_1 and t_3 are inconsistent with this *business rule*. The database designer might spot even more complex business rules. For example, if two sales reps sell the same product and have the same target, the one who has higher sales should not receive a lower bonus than the other. In Table 1.1, tuples t_5 and t_7 have the same value in columns Product and Target. Between those two, tuple t_5 has the highest value in Sales, so it should not have the lowest value in Bonus.

Table 1.1: A salesReps table that satisfies (traditional) integrity constraints.

	ID	Name	Address	City	ST	Zip	Product	Target	Sales	Salary	Bonus	SID
t_1	11	Ann Lee	8 Cornell	Palo Alto	CA	94306	Beer	$50000	$60000	$5000	$600	11
t_2	12	Dee Lee	8 Cornell	Palo Alto	CA	9430	Beer	$30000	$20000	$3000	$40	11
t_3	13	Elle Gray	2 Yale St	Palo Alto	CO	94306	Beer	$30000	$10000	$9000	$20	11
t_4	14	Ben Hill	3 Bowery	New York	NY	10012	Wine	$40000	$48000	$4000	$240	14
t_5	15	Amy King	8 3rd Ave	New York	NY	10003	Wine	$30000	$20000	$3000	$5	14
t_6	16	Ben King	8 3rd Ave	New York	NY	10003	Wine	$30000	$20000	$3000	$10	14
t_7	17	Abe Gray	2 8th Ave	New York	NY	10018	Wine	$30000	$10000	$3000	$10	14

No matter how fast a database management system can process queries, it is likely to return incorrect answers if the database contains inconsistencies. Here is an example of an SQL query that finds the sum of sales of the sales reps living in the state of California (CA):

```
1 select sum(Sales)
2 from   salesReps
3 where  ST = 'CA'
```

This query returns \$80000 because the `where` clause selects only tuples t_1 and t_2, but it should return \$90000 since tuple t_3 should also be selected, assuming that the values of city and zip code in tuple t_3 are correct and determine the value California (CA) for state. Now consider another query that finds the total amount of salaries paid to all sales reps:

```
1 select sum(Salary)
2 from   salesReps
```

This query returns \$30000. However, we should not trust this result either because of the inconsistency between the salaries in tuples t_1 and t_3. As we can see, even in small tables, there can be numerous data inconsistencies that lead to unreliable results. Of course, the level of inconsistency (and other data quality issues) in enterprise data can reach critical dimensions [6, 7].

Traditional integrity constraints—domain constraints, keys, and foreign keys—cannot identify the inconsistencies we saw in Table 1.1, which leads to the question of *how to define and enforce dependencies of higher expressive power*. The database textbook answer to this question is the concept of *assertions*; and *triggers* (or *active rules*) in *active databases* [8, 9, 10, 11]. Many major commercial database management systems do not support assertions, a piece of SQL that ensures a condition to be true. On the other hand, a couple of commercial database management systems provide some support for triggers, which is useful because triggers can check conditions, and thus, they generalize assertions.

The primary use of triggers is handling dependencies that cannot be expressed as the traditional integrity constraints: triggers signalize and rollback transactions having violations of integrity constraints [10, 12]. However, the injudicious use of triggers may lead to critical issues, other than the lack of data integrity. For years, experienced database researchers and practitioners have been expressing many concerns about triggers [10, 13, 12, 11, 14]. The lack of uniformity between database vendors, high maintainability costs, and low performance are among the most concerning pitfalls. The general advice is to use constraint mechanisms instead of triggers whenever possible [8, 10, 13, 12, 11, 14, 15].

1.1 PERSPECTIVES ON DEPENDENCIES

Constraints and dependencies are central concepts in relational database management systems as they concern the semantic integrity of relational data. The practical significance of these concepts has led all major database vendors to support built-in integrity constraints in their products. Besides, the formal foundation of dependencies is already quite solid, although less well-developed theories drive new efforts in theoretical research now and then [4, 16, 17, 18, 19].

The term constraint often refers to properties tied to database designs and requires enforcement, whereas the term dependency relates to properties of particular database instances that not necessarily require enforcement. For instance, the values in the column Name of salesReps

are all unique, thus, Name is a type of dependency known as *unique column combination*, or simply a *unique*. Notice, however, that defining Name as a primary key is a poor choice in database design since duplicate names are likely to happen. Although some dependencies might not require enforcement, dependencies, in general, are the primary vehicle for incorporating semantic properties into the relational model. Nevertheless, the terms constraints and dependencies widely appear as synonyms in the database literature [20, 4].

Dependencies started being a vivid topic in database research, as well as industry-based projects, soon after the proposal of the relational data model itself [2, 3]. Since then, the research on dependencies has produced numerous contributions that expand to multiple database contexts; naturally, since dependencies concern a broad topic. Recently the increasing demand for data of higher quality has motivated even further research on many types of dependencies.

We continue to discuss multiple perspectives on dependencies in the following.

1.1.1 Expressive power of dependencies

Different types of dependencies have different levels of expressive power, which means that some of them can restrict inconsistencies that others cannot. The higher the expressive power, the higher the complexity and, thus, the challenge in practical use. Some types of dependencies are computationally hard to handle. That is why the native support for dependencies in database management systems is somewhat limited—it is a trade-off between feasibility and expressiveness.

Dependencies. In this work, we focus on dependencies on single tables, sometimes called *intra-relation dependencies*. Other than unique column combinations, one of the most well-known examples of intra-relation dependencies is functional dependencies. Consider a relation schema R having instances r and two sets of columns of R, for instance, $X \subset \mathsf{R}$ and $Y \subset \mathsf{R}$. We denote $\mathsf{R}\colon X \rightarrow Y$ a functional dependency in a relation R, or simply $X \rightarrow Y$. This dependency states that the values of a tuple in X must *uniquely*, or *functionally*, determine the values of that tuple in Y. The following functional dependencies should hold in the salesReps instance in Table 1.1:

$$\mathsf{salesReps}\colon \mathsf{Address}, \mathsf{City}, \mathsf{ST} \rightarrow \mathsf{Zip},$$
$$\mathsf{salesReps}\colon \mathsf{Zip} \rightarrow \mathsf{City}, \mathsf{ST}.$$

The instance in Table 1.1 is inconsistent with the dependencies above because of tuples t_1,t_2 and t_3. As a result, any database maintaining this relation instance is also inconsistent since it is likely to produce incorrect answers. As we can see, data consistency is a concept related to sets of dependencies.

Due to historical and practical reasons, functional dependencies are one of the most well-studied dependencies in databases. Besides, there are various studies on *generalizations* of functional dependencies [21, 22, 23]. We review such generalizations, and other types of dependencies, in Chapter 2.

Relaxed dependencies. Production data is likely to contain errors and exceptions, even if it is in small numbers. A large table might contain only a few inconsistent tuples. A dependency might hold in part of the data alone. Besides, entities might appear multiple times in the database in various forms, for instance, "Ann Lee" and "Lee, Ann" as the same entity.

The general definitions of dependencies do not admit errors or exceptions in data. Dependencies following these strict definitions are sometimes called *exact* dependencies. The use of only exact dependencies can be impractical in many scenarios; thus, there are many studies on different forms to *relax* the canonical definitions of dependencies [23]. For example, *conditional functional dependencies* are a generalization of functional dependencies that specify conditions in which the dependencies hold; they are well-known in the data cleaning context [24, 25]. When we define a functional dependency, we expect that all tuples in the relation instance satisfy that dependency. On the other hand, when we define a conditional functional dependency, we assume that dependency to hold in a subset of tuples only, which are those tuples having the same (constant) pattern of values for some columns.

The following statement is an example of conditional functional dependency for salesReps:

$$\mathsf{salesReps}\colon ([\mathsf{Product} = \text{'Wine'}, \mathsf{Target}] \rightarrow [\mathsf{Salary}])\,.$$

This dependency specifies that all sales reps selling 'Wine' have their salaries determined by their targets. Table 1.1 is consistent with this dependency, although it would be inconsistent for the functional dependency counterpart $\mathsf{salesReps}\colon \mathsf{Product}, \mathsf{Target} \rightarrow \mathsf{Salary}$.

Conditional functional dependencies are just one of the examples of *relaxed dependencies*; there are many others [23], which we discuss in Chapter 2.

Denial constraints. The reader might have noticed that the dependencies presented so far still cannot identify all inconsistencies in salesReps, even if we consider all of them together. That is because these various dependencies fall short of adequate expressive power. To directly address this sort of shortcomings, a large part of this thesis regards *denial constraints*, a type of dependency of high expressive power that can also incorporate relaxation definitions.

Denial constraints use relationships between predicates to specify inconsistent states of column values. We give formal definitions in subsequent chapters, but for now we express a denial constraint φ as follows:

$$\varphi\colon \forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}, \neg(\mathsf{p}_1 \wedge \ldots \wedge \mathsf{p}_m),$$

where t_x and t_y are tuples of table r; and p_i are predicates drawn from the schema R of r. A predicate p has one of the forms $\mathsf{t}_x.\mathsf{A}\ \mathsf{o}\ \mathsf{t}_y.\mathsf{B}$ or $\mathsf{t}_x.\mathsf{A}\ \mathsf{o}\ \mathsf{c}$, where A,B are columns of R (A and B can refer to the same column); o is an operator in $\mathsf{O} = \{=, \neq, <, \leq, >, \geq\}$; and c is a constant drawn from the domain of column A. A denial constraint φ in the above form states that there

cannot exist a pair of tuples t_x, t_y in table r satisfying all predicates of φ simultaneously; if there exists such a t_x, t_y , then r is inconsistent with the denial constraint φ.

Table 1.2 shows the denial constraints that model the dependencies discussed so far (the tuple identifiers are omitted). For example, the denial constraint φ_1 states that if any two sales reps have the same values in $\{\text{Address}, \text{City}, \text{ST}\}$, then they must have the same value in $\{\text{Zip}\}$. In other words, if a pair of tuples t_x, t_y of salesReps satisfies the predicates $t_x.\text{Address} = t_y.\text{Address}$, $t_x.\text{City} = t_y.\text{City}$ and $t_x.\text{ST} = t_y.\text{ST}$ simultaneously, then it cannot satisfy the predicate $t_x.\text{Zip} \neq t_y.\text{Zip}$. Similar interpretations goes for the remaining denial constraints.

Table 1.2: Denial constraints that capture the data inconsistencies in salesReps.

Semantics	Denial constraint
Address, city, and state determine zip.	$\varphi_1 : \neg(t_x.\text{Address} = t_y.\text{Address} \wedge t_x.\text{City} = t_y.\text{City} \wedge t_x.\text{ST} = t_y.\text{ST} \wedge t_x.\text{Zip} \neq t_y.\text{Zip})$
Zip determines state.	$\varphi_2 : \neg(t_x.\text{Zip} = t_y.\text{Zip} \wedge t_x.\text{State} \neq t_y.\text{State})$
Sales reps cannot earn higher salaries than their supervisors.	$\varphi_3 : \neg(t_x.\text{SID} = t_y.\text{ID} \wedge t_x.\text{Salary} > t_y.\text{Salary})$
Targets determine the salaries of all sales reps selling 'Wine'.	$\varphi_4 : \neg(t_x.\text{Product} = t_y.\text{Product} \wedge t_x.\text{Product} = \text{'Wine'} \wedge t_x.\text{Target} = t_y.\text{Target} \wedge t_x.\text{Salary} \neq t_y.\text{Salary})$
If two sales reps sell the same product and have the same target, the one who has higher sales should not receive a lower bonus than the other.	$\varphi_5 : \neg(t_x.\text{Product} = t_y.\text{Product} \wedge t_x.\text{Target} = t_y.\text{Target} \wedge t_x.\text{Sales} > t_y.\text{Sales} \wedge t_x.\text{Bonus} < t_y.\text{Bonus})$

The research questions on types of dependencies of higher expressive power, such as denial constraints, are challenging. Nonetheless, the answers to such questions pursue the development of adequate support for dependencies that can cover a broad range of data inconsistencies.

1.1.2 Discovery of dependencies

Relational database design and maintenance is a complex process that requires, among other tasks, defining sets of dependencies. One option is to delegate the task to database designers with adequate expertise in the domain of the application. Although this option may work for small databases and simple types of dependencies, it may become infeasible in other scenarios. Database designers with enough expertise might not be conveniently available. Even when experts are around, the manual design of dependencies is time-consuming as experts must keep the dependencies up-to-date with the semantics of data and application, which is continually evolving. Besides, the higher the expressive power of a dependency language, the higher the

complexity in the design of dependencies. Finally, the number of possible dependency candidates is usually too large for manual validation, even in small datasets.

The alternative to the manual design of constrains is the automatic discovery of dependencies using data [21]. In a nutshell, the dependency discovery problem is to find the set of dependencies, in a particular language, that holds in a specific table. The problem comes under the umbrella of *data profiling*: the set of activities to gather statistical and structural properties, i.e., *metadata*, about datasets [22]. In general, the challenges in the discovery problem are as following:

***Enumeration* and *checking* of dependency candidates.** In theory, the number of possible dependencies in a table is exponential in the number of columns in the schema. As a consequence, discovery algorithms regard combinatorial problems having, in the worst case, exponential time complexity. The higher the expressive power of a dependency language, the higher the number of candidates and, thus, the harder the enumeration and checking of dependency candidates in an efficient manner.

Use of inconsistent data to discover consistent dependencies. The discovery of dependencies from data might return non-reliable results because the available data might be inconsistent. Even if the discovery relies on data having only a small amount of inconsistency, the dependencies identified are likely inconsistent themselves. The inclusion of relaxation definitions into the discovery problem is a well-known way to circumvent the problem. However, the discovery of relaxed dependencies is harder than the discovery of exact dependencies because the former problem cannot use many optimizations that drastically reduce search spaces.

Unreliable results. The results of dependency discovery algorithms might hold only accidentally. Whether the discovery relies on consistent data or not, the number of results is usually huge, and, in all likelihood, not all results are equally useful. A large part of the results might be only residuals of overfitting, and only a few may support the improvement of data integrity or any other data management task.

The discovery of basic types of dependencies has long been studied [26]. In [27], the authors compare implementation details and experimental evaluation of seven algorithms for the discovery of functional dependencies. Since this publication, several additional papers focusing on functional dependency discovery were published [28, 29, 30, 31]. In contrast, the discovery of more complex types of dependencies is in the early stages of development, still with a limited number of contributions. For example, by the time we started this research project, there was only one publication on the discovery of denial constraints, called FASTDC [32].

One of the contributions in this thesis is DCFINDER [41], an efficient algorithm to discover both exact and relaxed denial constraints. DCFINDER is designed to overcome many of the pitfalls observed in previous solutions for the discovery of denial constraints. Also, this thesis presents a novel technique to help DCFINDER avoid returning unreliable results [42]. The main goal of this technique is to select denial constraints that can identify errors in datasets. This thesis

also investigates the problem of discovering functional dependencies for query optimization and presents FDSEL [44]. The tool selects a set of functional dependencies from data profiles, which can be used in query rewritings and benefit query executions.

1.1.3 Violations of dependencies

A *dependency violation* is a tuple (or set of tuples) having values that do not agree with the semantics of the dependency. That way, data inconsistencies emerge as violations of the dependencies defined for the database.

Databases may become inconsistent due to different reasons. For example, a poorly designed database is likely to store inconsistent data. A database management system and its applications may not have enough mechanisms to ensure adequate data consistency. Besides, in data integration scenarios, multiple different databases have various perspectives on data consistency. Choosing a global definition of consistency is already hard, and so is matching all the various data to this definition [33]. In general, many production databases are subject to data inconsistencies at some point.

Detection of dependency violations. Knowing to which extend inconsistencies permeates a database is the first step towards producing better-quality query answers; therefore, the detection of dependency violations is vital. In data cleaning pipelines, nothing can be done before the detection step. Even if fixing inconsistencies is not possible, users surely need to be aware of the inconsistencies so they can avoid poor decision-making.

The most straightforward way to detect a dependency violation is to enumerate the necessary combination of tuples, and then check whether each combination complies with the dependency or not. For example, a naive approach for detecting the violations of the denial constraints in Table 1.2 would enumerate and check every pair of tuples in the table against all predicates of each denial constraint. Of course, this approach is impractical for large datasets since it has a quadratic time complexity in the number of tuples.

An alternative to the naive approach is to translate dependencies into SQL queries and then ask a database management system to find the violations. Although the use of database management systems is practical, it has two critical performance drawbacks. The performance varies significantly from system to system, and, worst yet, it is usually not robust against different types of dependencies. For the same dataset, a database management system may perform well for a given dependency but perform poorly for another (we investigate this issue in Chapter 5).

Most of the recently presented data cleaning systems use database management systems to detect violations of data dependencies. Still, their experimental evaluations are quite limited, as they explore mostly simple dependencies (e.g., functional dependencies) and small datasets [34, 35, 36, 24]. In many real-world scenarios, however, data cleaning (and other data management tasks) has to deal with large datasets and complex dependencies such as denial constraints. Thus, there is a need for efficient techniques to detect violations of dependencies of various types.

This thesis also presents VIOFINDER [43], a system for efficient detection of violations of data dependencies. VIOFINDER includes many novel concepts and algorithms that enable the tool to outperform three commercial data management systems and another dependency-based tool in several scenarios.

Handling of dependency violations. There are two primary courses of action for handling data inconsistencies. The first, *consistent query answer*, allows both consistent (clean) and inconsistent (dirty) data to coexist in the database [37]. When applications submit queries to the database, a solution for consistent query answering must compute consistent views of the data at runtime—these views are called *repairs*. Then, from these repairs, it needs to determine which ones are the best to retrieve the (consistent) answers to the initial query. The second course of action is data repairing [38]. The idea is to combine the inconsistent database with a series of data updates to produce a new database (also called repair) that satisfies the constraints in the database, and therefore, is free from inconsistencies. The changes to the inconsistent database must be as minimal as possible.

1.1.4 Applications of dependencies

Dependencies incorporate semantics into the relational model that enable the support or improvement of data quality, query performance, and database design.

Data quality. Data consistency is a central dimension of data quality [39, 40]. Due to the increase in interest for high-quality data, the most investigated use of dependencies in the last years has been data cleaning, or related subjects aiming at increasing data quality. As discussed earlier, database management systems automatically check update operations for compliance with types of integrity constraints of limited expressive power. Unfortunately, that is not enough to ensure data with high standards of quality. Nonetheless, some techniques can help in improving data consistency, and thus, data quality in general. We review many of them in Chapter 2. This thesis makes contributions on two dimensions of data consistency: discovery of dependencies; and detection of dependency violations.

Query performance. Databases that satisfy specific dependencies, for example, functional dependencies, may benefit from an extended search space of possible query execution plans. Query optimizers can leverage dependencies to determine better query execution plans or rewrite queries into semantically equivalent ones that result in better performance. This thesis makes a contribution that combines results from an automatic discovery of functional dependencies with the application of dependencies in query optimization.

Database design. Dependencies are the fundamentals of relational database design. Good quality designs avoid table schemas that allow data anomalies to persist in the database. The data *normalization* process is to decompose a poorly design table schemas into well-defined ones that satisfy *normal forms*, which are based on dependencies (in general, functional dependencies and a few variants). We provide further references on this topic in Chapter 2.

1.2 SUMMARY OF CONTRIBUTIONS

The research on data dependencies is vibrant, but at the same time, challenging. Contributions on the field have numerous applications in various data management aspects. The contributions of this thesis cover four primary dimensions, summarized as follows.

A novel algorithm for the discovery of denial constraints [41]. The alternative to designing denial constraints by hand is automatically discovering denial constraints from data. Unfortunately, this alternative is computationally expensive due to the vast search space derived from the number of predicates that can form denial constraints. To tackle this challenging task, we present a novel algorithm, DCFINDER. It combines data structures called position list indexes, bitwise operations, and optimizations based on predicate selectivity to validate denial constraint candidates efficiently. Because the available data often contain errors, the design of DCFINDER algorithm focuses on the discovery of relaxed denial constraints. Our experimental evaluation uses real and synthetic datasets and shows that DCFINDER outperforms previous existing algorithms for the discovery of relaxed denial constraints.

A novel technique to focus the dependency discovery in denial constraints useful for data cleaning [42]. In the traditional approach to the discovery of dependencies, the results are as reliable as the data used to produce them. Having problematic data is often involuntary; thus, the discovery should be able to accommodate potential data errors. Besides, the number of discovered results grows exponentially with the number of columns in the table. Even if we discover dependencies from correct data, many results may hold only by chance, i.e., they are spurious. We propose a method that uses statistical evidence of the tuples of a dataset to focus the discovery of denial constraints in dependencies of interest. Our method sets DCFINDER so that it can find denial constraints appropriate for data cleaning, even if the dataset contains errors. Our experiments with real data show that the identified denial constraints point, with high precision and recall, to inconsistencies in the input data.

A novel system to detect violations of denial constraints [43]. Dependencies and their violations can reveal errors in data. Several data cleaning systems use database management systems to detect violations of data dependencies. While this approach is efficient for some kinds of data dependencies (e.g., key dependencies), it is likely to fall short of satisfactory performance for more complex ones, such as some forms of denial constraints. We propose a novel system to detect violations of denial constraints efficiently. We describe its execution model, which operates on compressed blocks of tuples at-a-time, and we present various algorithms that take advantage of the predicate form in denial constraints to provide efficient code patterns. Our experimental evaluation includes comparisons with different approaches; real-world and synthetic data; and various kinds of denial constraints. It shows that our system is up to three orders-of-magnitude faster than the other solutions, especially for datasets with a large number of tuples and denial constraints that identify a large number of violations.

Novel techniques to detect functional dependencies appropriate for query optimization [44]. We present a system for automatic query optimization based on data dependencies. By formulating query transformations, it can revise the number of processed rows, with a direct impact on performance. The goal is to optimize query execution in cases where the database is denormalized or have lost dependencies in the design. We rely on the automatic discovery of dependencies, but to avoid optimizing for spurious dependencies, we focus on dependencies matching the current queries in the pipe (i.e., the *workload*). Initially, we use a state-of-the-art algorithm to discover the set of functional dependencies holding in the datasets. Then, our *focused dependency selector* uses the available workload information to choose exemplars from the set of the discovered functional dependencies that are appropriate for query optimization. That eliminates any manual interaction. The selected dependencies exhibit statistical properties that resemble those of the initial set of dependencies; therefore, they serve as a semantical summary of the dependencies. We use well-known techniques for query optimization with the selected dependencies. In the best-case scenario of our experimental evaluation, our system can reduce query response time by more than one order of magnitude using join elimination for a real-world database.

1.3 THESIS OUTLINE

The outline of the remainder of this thesis is as follows. Chapter 2 provides background on dependencies; and discusses many works related to our primary contributions. The next four chapters are based on the works we published during the development of this thesis (see Appendix A). Chapter 3 presents our algorithm for the discovery of denial constraints: DCFINDER. Then, Chapter 4 proceeds and presents our solution to detect denial constraints appropriate for data cleaning. Chapter 5 describes a novel system for the detection of violations of denial constraints. Chapter 6 presents our tool for detecting functional dependencies appropriate for query optimization. Finally, Chapter 7 concludes this study with a discussion on closing thoughts and topics for future work.

Chapter 2

Background

In this chapter, we present the necessary notations and describe numerous concepts associated with this thesis. Besides, we discuss several research problems and works related to the primary contributions of this thesis.

2.1 BASIC NOTATIONS AND CONVENTIONS

We consider relation instances r, or tables r for short, of relation schemas $\mathsf{R}(\mathsf{A}_1, ..., \mathsf{A}_n)$. The possible values of each column (or attribute) $\mathsf{A}_i \in \mathsf{R}$ are drawn from its domain $dom(\mathsf{A}_i)$. Each tuple t of r is an element of the Cartesian product $dom(\mathsf{A}_1) \times \ldots \times dom(\mathsf{A}_n)$. When referencing the tuples in tables, we consider the position of tuples within the table as tuple identifiers (also called *offset*); see Table 1.1. We use X and Y to denote sets of columns, and we use A to reference each column in X, and B to reference each column in Y, that is, $\mathsf{A} \in X$ and $\mathsf{B} \in Y$. We denote the projection of a tuple on a set of columns (or single column) using brackets, for example, $\mathsf{t}[X]$ or $\mathsf{t}[\mathsf{A}]$.

Let $\mathsf{O} = \{=, \neq, <, \leq, >, \geq\}$ be a set of built-in operators of the database. Predicates p are comparison expressions of the form $\mathsf{t}_x.\mathsf{A}_i \ \mathsf{o} \ \mathsf{t}_y.\mathsf{A}_j$ or $\mathsf{t}_x.\mathsf{A}_i \ \mathsf{o} \ \mathsf{c}$, where columns $\mathsf{A}_i, \mathsf{A}_j \in \mathsf{R}$; tuples $\mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}$; operator $\mathsf{o} \in \mathsf{O}$; and c is a constant drawn from $dom(\mathsf{A}_i)$. Predicates can compare two tuples for the same column, so the two columns in a predicate can be the same ($i = j$). For convenience, we sometimes also use A and B to refer to the columns in predicates. The above predicate notation is useful in expressing denial constraints (as we can see in Table 1.1). Besides, it is close to statements often seen in the where clauses of standard SQL queries. Given a predicate p, we denote $\mathsf{p}.\mathsf{A}_\mathsf{i}$ the column in its left-hand-side; $\mathsf{p}.\mathsf{A}_\mathsf{j}$ the column in its right-hand-side; and $\mathsf{p}.\mathsf{o}$ its operator. Given any two predicates p_1 and p_2, we write $\mathsf{p}_1 \sim \mathsf{p}_2$ to say that the columns $\mathsf{p}_1.\mathsf{A}_i = \mathsf{p}_2.\mathsf{A}_i$ and $\mathsf{p}_1.\mathsf{A}_j = \mathsf{p}_2.\mathsf{A}_j$, and $\mathsf{p}_1 \not\sim \mathsf{p}_2$ to say otherwise.

Figure 2.1 shows the implication of each operator $\mathsf{o} \in \mathsf{O}$. A predicate $\mathsf{p}_1 : \mathsf{A}_i \ \mathsf{o} \ \mathsf{A}_j$ implies every predicate $\mathsf{p}_2 : \mathsf{A}_i \ \mathsf{o}' \ \mathsf{A}_j$, where $\mathsf{o}' \in \mathsf{o}_\Rightarrow$. If predicate p_1 is true, then every implication p_2 is true. We denote $imp(\mathsf{p})$ the set of predicates implied by p. Figure 2.1 also shows the logical complement $\bar{\mathsf{o}}$ of each operator. The complement of a predicate $\mathsf{p} : \mathsf{t}_x.\mathsf{A}_i \ \mathsf{o} \ \mathsf{t}_y.\mathsf{A}_j$ is the predicate

$\bar{p} : t_x.A_i\ \bar{o}\ t_y.A_j$, where $\bar{o}$ is the logical complement (or negation) of operator o. If predicate p is true, then $\bar{p}$ is false.

Operator (o)	$=$	$\neq$	$<$	$\leq$	$>$	$\geq$
Implication ($o_{\Rightarrow}$)	$=, \leq, \geq$	$\neq$	$<, \leq, \neq$	$\leq$	$>, \geq, \neq$	$\geq$
Negation ($\bar{o}$)	$\neq$	$=$	$\geq$	$>$	$\leq$	$<$

Figure 2.1: Implication and complement of built-in operators of the database.

2.2 DEPENDENCIES

Database constraints are commonly expressed as dependencies that define a semantic property on a column or group of columns. Once defined, database constraints must be satisfied by any database instance. We can explicitly express some types of dependencies as constraints at the schema level using a data definition language—these constraints are sometimes called *schema-based constraints* or *explicit constraints.*

As we saw in Chapter 1, the basic integrity constraint framework of most commercial database management systems cannot express many critical types of semantic properties. The alternative then is to use dependencies of adequate expressive power to capture such properties. Dependencies have a less strict definition than constraints. A dependency is a property on a column or group of columns that apply to particular instances of the database. We can choose a dependency to be enforced as a constraint. If the database management system cannot implement this constraint, then we need to implement it using other means.

The initial studies on dependencies started shortly after the proposal of the relational model. Their primary motivation was mainly database design, but nowadays, dependencies are a fundamental part of various data management contexts. The research on dependencies contributed to a variety of dependency languages for defining the semantics of relational databases [45, 46, 47, 4]. Most dependency languages can be expressed using first-order logic sentences, naturally, as dependencies can be seen as semantic sentences about relations. In practice, however, dependency languages are restricted to find a balance between expressive power and language complexity [20, 48]. For a general perspective on dependency theory, we refer the reader to the following comprehensive study [20], and books [49, 50, 4].

Static analysis. There are essential theoretical problems regarding the dependency theory. Of course, the results for these problems vary according to dependency languages. The higher the expressive power of a dependency language, the harder its complexity results [20, 4, 1]. We describe two of these essential problems, also referred to as *static analysis* of a dependency language, as they are central in developing practical solutions based on dependencies [48].

The first problem is about *satisfiability*. Given a set Σ of dependencies, expressed in a dependency language $\mathscr{L}$ and defined on a relation schema R, the satisfiability problem for $\mathscr{L}$ is

about whether there exists a nonempty relation instance r of R that satisfies every dependency φ in Σ. We write $\mathsf{r} \models \varphi$ to say that r satisfies φ, and $\mathsf{r} \not\models \varphi$ to say otherwise. The satisfiability problem regards the consistency of the dependencies themselves. If the set Σ cannot be satisfied by any relation instance, then using Σ to validate data becomes pointless.

The second problem is about *implication*. Consider a set Σ of dependencies and a dependency φ, expressed in a dependency language $\mathscr{L}$ and defined on a relation schema R. The implication problem for $\mathscr{L}$ is about deciding whether Σ implies φ. This implication is true if $\mathsf{r} \models \Sigma$, then $\mathsf{r} \models \varphi$, for every relation instance r of R. We write $\Sigma \models \varphi$ to say that Σ implies φ, and $\Sigma \not\models \varphi$ to say otherwise. Notice that the dependency φ is redundant if $\Sigma \models \varphi$. Avoiding such redundancies helps in several practical problems. For example, in the detection of dependency violations, if we have $\Sigma \models \varphi$, then the violations for φ are contained in the violations for Σ. Therefore, there is a great interest in the development of algorithms for determining the implication of dependencies.

Another perspective on the implication problem is the study of *inference rules* as a mechanism to determine logical implication—a well-known example is Armstrong's Axioms for functional dependencies. An important property from inference rules is that if there exists a finite set of inference rules for a dependency language, then there exists an algorithm for determining the logical implication [4].

Dependencies considered in this work. We consider *state (or static) dependencies*. Those are dependencies that define properties for the states of a database. Another type of dependency is *dynamic (or transition) dependencies*, which defines properties for database value changes: for example, "the age of a person can only increase." Dynamic dependencies are out of the scope of this work, but the reader can find pointers on the subject in [51]. Also, we restrict this study to dependencies involving only single tables, sometimes called *intra-relation dependencies*. The study of dependencies involving multiple tables at a time is out of the scope of this work—the reader can find material on the subject in [52, 53, 4].

In the following, we present the fundamentals and notations for denial constraints and functional dependencies, as they are related to our main contributions. Also, we discuss some other types of dependencies related to this work. For examples, we consider the relation instance salesReps once more—Table 1.1 in Chapter 1.

2.2.1 Denial constraints

Denial constraints are one of the most general types of intra-relation dependencies discussed in database literature since they have high expressive power and generalize several different types of dependencies [32, 54, 1]. They are a universally quantified first-order logic formalism. Each denial constraint expresses a set of relational predicates that specify constraints for inconsistent combinations of column values. Any tuple, or set of tuples, that disagrees with these constraints is a denial constraint violation that reflects inconsistencies in the database.

A denial constraint can involve multiple tuples; however, denial constraints involving more than two tuples are less likely to represent useful business rules. Considering an unlimited number of tuples in each constraint leads to serious complexity and feasibility issues, at the cost of controversial gains in expressive power. In general, denial constraints involving at most two tuples suffice to represent most of the constraints required in practice. Besides, this class of denial constraints can already generalize many other essential types of dependencies and represent a vast range of complex business rules. Therefore, in this work, we consider denial constraints involving at most two tuples—related work apply the same restriction [32, 55, 34]. We express the universal quantifiers for denial constraints involving at most two tuples as $\forall t_x, t_y$, and we express denial constraints using sets of predicates of the form defined in Section 2.1.

Definition 1 (Denial Constraint)**.** A denial constraint φ over a relation instance r is a statement of the form

$$\varphi : \forall t_x, t_y \in r, \neg(p_1 \wedge \ldots \wedge p_m)$$

where φ is *satisfied* by r if and only if for any tuple pair $t_x, t_y \in r$ at least one of the predicates $p_1, \ldots, p_m$ is false. In other words, the denial constraint φ does not hold if there exists any tuple pair in r that satisfies all the predicates of φ.

We write $t_x, t_y \models \varphi$ to say that a tuple pair t_x, t_y satisfies φ, and $t_x, t_y \not\models \varphi$ to say otherwise. We say that a denial constraint φ_1 implies another denial constraint φ_2, written as $\varphi_1 \models \varphi_2$, if for every relation instance r, the statement $r \models \varphi_1$ implies $r \models \varphi_2$.

A denial constraint φ is called *trivial* if it is satisfied by any relation instance. For example, the denial constraint:

$$\varphi_6 : \forall t_x, t_y \in \mathsf{salesReps}, \neg(t_x.\mathsf{Name} = t_y.\mathsf{Name} \wedge t_x.\mathsf{Name} \neq t_y.\mathsf{Name})$$

is trivial, since it is valid in any instance of salesReps—no pair of tuples can have equal names and different names at the same time. The *symmetric* denial constraint φ_2 of a denial constraint φ_1 is given by swapping the tuple identifiers t_x and t_y in the predicates of φ_1. For example, the denial constraint:

$$\varphi_7 : \forall t_x, t_y \in \mathsf{salesReps}, \neg(t_x.\mathsf{ID} = t_y.\mathsf{SID} \wedge t_x.\mathsf{Salary} < t_y.\mathsf{Salary})$$

is symmetric to the denial constraint φ_3 in Table 1.2. Notice that if denial constraints φ_1 and φ_2 are symmetric, then $\varphi_1 \models \varphi_2$, and vice versa.

A denial constraint φ_1 is *minimal* if there does not exist a φ_2 such that both φ_1 and φ_2 are satisfied by r and the predicates of φ_2 are a subset of φ_1. In other words, a denial constraint φ_1 is not minimal if it is a generalization of another denial constraint φ_2. For example, the denial constraint:

$$\varphi_8 : \forall t_x, t_y \in \mathsf{salesReps}, \neg(t_x.\mathsf{Zip} = t_y.\mathsf{Zip} \wedge t_x.\mathsf{City} = t_y.\mathsf{City} \wedge t_x.\mathsf{State} \neq t_y.\mathsf{State})$$

is not minimal in the relation instance salesReps, since the set of predicates of the denial constraint φ_2 in Table 1.2 is a subset of the predicates of denial constraint φ_8—the predicate $t_x.\mathsf{City} = t_y.\mathsf{City}$ in φ_8 is not necessary.

We can form predicates by combining the columns of a relation schema: with each other, or with the values in their domains. Besides, we can use different built-in operators and derive predicates of various forms. Then, we can combine these predicates in many different ways to represent numerous types of denial constraints. This great variety of possibilities illustrates the high expressive power of denial constraints. The denial constraints in Table 1.2 are simple examples of how we can use the formalism to express complex business rules (denial constraints φ_3 and φ_5), or other types of dependencies (denial constraints φ_1 and φ_2 as functional dependencies, and denial constraint φ_4 as a conditional functional dependency).

Static analysis. The satisfiability problem for denial constraints has not been established yet [1]. However, it has already been shown that the problem is NP-complete for some types of dependencies subsumed into denial constraints, for example, conditional functional dependencies [24, 1]. The implication problem for denial constraints is coNP-complete [56]. In this matter, a sound, but not complete, inference system for denial constraints is presented in [32]. The soundness is in the sense that every denial constraint inferred from a set of denial constraints using the inference system is indeed a denial constraint implied by that set of denial constraints. The completeness is in the sense that there might exist denial constraints derived from a set of denial constraints that cannot be inferred using the inference system. Still, the proposed inference system helps in the discovery of denial constraints as it enables pruning of the search space. Also, because implied and trivial denial constraints are removed, the system may promote a reduction in the number of discovered results. The inference system for denial constraints includes three rules (here, we also use q and s to refer to predicates, for convenience) [32]:

(Triviality). $\forall \mathsf{p}_i, \mathsf{p}_j$ if $\bar{\mathsf{p}}_i \in imp(\mathsf{p}_j)$, then $\neg(\mathsf{p}_i \wedge \mathsf{p}_j)$ is a trivial denial constraint.

(Augmentation). If $\neg(\mathsf{p}_i \wedge \ldots \wedge \mathsf{p}_n)$ is a valid denial constraint, then $\neg(\mathsf{p}_i \wedge \ldots \wedge \mathsf{p}_n \wedge \mathsf{q})$ is also a valid denial constraint.

(Transitivity). If $\neg(\mathsf{p}_i \wedge \ldots \wedge \mathsf{p}_n \wedge \mathsf{q}_1)$ and $\neg(\mathsf{s}_i \wedge \ldots \wedge \mathsf{s}_m \wedge \mathsf{q}_2)$ are valid denial constraints, and $\mathsf{q}_2 \in imp(\bar{\mathsf{q}}_1)$, then $\neg(\mathsf{p}_i \wedge \ldots \wedge \mathsf{p}_n \wedge \mathsf{s}_i \wedge \ldots \wedge \mathsf{s}_m)$ is also a valid denial constraint.

The triviality rule specifies that if a denial constraint is trivial if it contains two predicates that cannot be true at the same time. The Augmentation rule concerns the addition of unnecessary predicates to a denial constraint: if a denial constraint is already valid, then adding another predicate to it results in another valid denial constraint. Finally, the transitivity rule concerns two denial constraints and two predicates (one predicate in each denial constraint) and assumes that these two predicates cannot be true simultaneously. Then, the combination of these two denial constraints having those two predicates removed results in a valid denial constraint.

2.2.2 Functional dependencies

One of the most common dependencies in database literature is arguably functional dependencies. We briefly discussed them in Chapter 1. In the follow, we give their definition.

Definition 2 (Functional dependency)**.** A functional dependency $f\colon X \to Y$ states that the values of a tuple in X must uniquely or functionally determine the values of that tuple in Y. A relation instance r satisfies f if for all pair of tuples, $\mathsf{t}_x, \mathsf{t}_y$, in r the following condition holds:

$$\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}\colon \mathsf{t}_x[X] = \mathsf{t}_y[X] \implies \mathsf{t}_x[Y] = \mathsf{t}_y[Y].$$

The right-hand side Y of f is functionally determined by the left-hand side X. We denote $f.lhs$ the left-hand side, and $f.rhs$ the right-hand side of a functional dependency f. A functional dependency is non-trivial if it does not have any redundant attribute (i.e., $X \not\subseteq Y$), and it is minimal if there exists no set Z such that $(X - Z) \to Y$ is also a valid functional dependency. We can decompose a functional dependency f into multiple functional dependencies using each column in the right-hand side of f. For example, consider a functional dependency $X \to Y$ where $Y = \{\mathsf{B}_1, \mathsf{B}_2\}$. The two functional dependencies $X \to \mathsf{B}_1$ and $X \to \mathsf{B}_2$ are equivalent to the single functional dependency $X \to Y$.

We can express functional dependencies using denial constraint notation because functional dependencies are subsumed into denial constraints. To do so, we can transform the implication in the definition of functional dependencies into conjunctions. Consider a functional dependency $X \to \mathsf{B}$; then we have the following implication:

$$\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}\colon \mathsf{t}_x[X] = \mathsf{t}_y[X] \implies \mathsf{t}_x[\mathsf{B}] = \mathsf{t}_y[\mathsf{B}].$$

First, we can write the projections as predicates, as follows:

$$\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}\colon \bigwedge_{\mathsf{A} \in X} \mathsf{t}_x.\mathsf{A} = \mathsf{t}_y.\mathsf{A} \implies \mathsf{t}_x.\mathsf{B} = \mathsf{t}_y.\mathsf{B}.$$

We have, from De Morgan's laws, that the negation of a conjunction $\bigwedge_{\mathsf{A} \in X} \mathsf{t}_x.\mathsf{A} = \mathsf{t}_y.\mathsf{A}$ is the disjunction of negations $\bigvee_{\mathsf{A} \in X} \mathsf{t}_x.\mathsf{A} \neq \mathsf{t}_y.\mathsf{A}$. Thus, we can apply the material implication rule to replace the above implication with the following disjunctions:

$$\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}\colon \bigvee_{\mathsf{A} \in X} \mathsf{t}_x.\mathsf{A} \neq \mathsf{t}_y.\mathsf{A} \vee \mathsf{t}_x.\mathsf{B} = \mathsf{t}_y.\mathsf{B}.$$

Still considering De Morgan's laws, we can transform the formula above into an equivalent denial by negating the complement of the above sentence as follows:

$$\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}\colon \neg \left(\bigwedge_{\mathsf{A} \in X} \mathsf{t}_x.\mathsf{A} = \mathsf{t}_y.\mathsf{A} \wedge \mathsf{t}_x.\mathsf{B} \neq \mathsf{t}_y.\mathsf{B} \right).$$

This is a valid denial constraint representation of the functional dependency $X \rightarrow \mathsf{B}$.

Static analysis. Any set of functional dependencies is satisfiable, and the implication problem for denial constraints is in linear time [4, 57]. The inference rules for functional dependencies, known as Armstrong's Axioms, are covered in most database textbooks, and they have inspired the inference system for denial constraints we saw before. For convenience, we reproduce the Axioms here:

(Reflexivity). If $X \subseteq Y$, then $X \rightarrow Y$

(Augmentation). If $X \subseteq Y$, then $XZ \rightarrow YZ$, where Z is also a set of columns. Here, XZ and YZ represent unions of sets of columns.

(Transitivity). If $X \subseteq Y$, and $Y \subseteq Z$ then $X \rightarrow Z$

2.2.3 Unique column combinations

A unique column combination is a set of columns for which every tuple in a relation instance has unique values. In other words, a set of columns X is a unique in r if $\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}$ and $x \neq y$, then $\mathsf{t}_x[X] \neq \mathsf{t}_y[X]$. Unique column combination are sometimes called *uniques*, *uniqueness constraints, candidate keys, or key dependencies*. Notice that uniques are a particular case of functional dependencies, as the set of column X determines all columns of the relation, that is, $X \rightarrow \mathsf{R}$. As two examples, column Name and column ID are valid uniques in salesReps—however, common sense says that the column ID works better as a primary key than Name.

We can apply a few transformation rules, in a similar way we did for functional dependencies, to convert a unique into logically equivalent expressions that represent a denial constraint. Thus, we can represent a unique X as a denial constraint as follows:

$$\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r} \colon \neg \left(\bigwedge_{\mathsf{A} \in X} \mathsf{t}_x.\mathsf{A} = \mathsf{t}_y.\mathsf{A} \right).$$

Static analysis. Unique column combinations are subsumed into functional dependencies; thus, it is possible to use the results of the static analysis of functional dependencies.

2.2.4 Order dependencies

Order dependencies specify relationships of order (sort) between the columns of a relation schema. The definition of order dependencies is based on the semantic adopted to order tuples; there are two variants: *pointwise ordering* and *lexicographical ordering* [58]. Consider a tuple t_x, having values $(\mathsf{t}_x[\mathsf{A}_1], \ldots, \mathsf{t}_x[\mathsf{A}_n])$, and a tuple t_y, having values $(\mathsf{t}_y[\mathsf{A}_1], \ldots, \mathsf{t}_y[\mathsf{A}_n])$. The pointwise ordering specifies that for $(\mathsf{t}_x[\mathsf{A}_1], \ldots, \mathsf{t}_x[\mathsf{A}_n]) < (\mathsf{t}_y[\mathsf{A}_1], \ldots, \mathsf{t}_y[\mathsf{A}_n])$ to be true, then $\mathsf{t}_x[\mathsf{A}_i] \leq \mathsf{t}_y[\mathsf{A}_i]$, for all $1 \leq i \leq n$. On the other hand, the lexicographical ordering specifies

that $(\mathsf{t}_x[\mathsf{A}_1],\ldots,\mathsf{t}_x[\mathsf{A}_n]) < (\mathsf{t}_y[\mathsf{A}_1],\ldots,\mathsf{t}_y[\mathsf{A}_n])$ is true if there exists some $i \geq 1$ such that $\mathsf{t}_x[\mathsf{A}_i] < \mathsf{t}_y[\mathsf{A}_i]$ and, for each $j < i$, $\mathsf{t}_x[\mathsf{A}_j] = \mathsf{t}_y[\mathsf{A}_j]$.

Pointwise order dependencies strictly generalize lexicographical order dependencies. Thus, there exists a mapping of any lexicographical order dependency into a set of pointwise order dependencies, as it is proven in [59]. The definition of pointwise order dependency, as it is given in [59], is based on *order conditions*, which are marked columns A° in which $\mathsf{o} \in \{=, <, \leq, >, \geq\}$—mind that the operator $\neq$ is not included here, differently than the operators in denial constraints. For this definition, let us consider that the sets X and Y are sets of order conditions instead of sets of standard columns. A pointwise order dependency $X \rightsquigarrow Y$ is valid if the value order in each marked column of X implies a value order in each column of Y. That is to say, a relation instance r satisfies an order dependency $X \rightsquigarrow Y$ if, for any pair of tuples t_x and t_y in r, the following holds: for each order condition $\mathsf{A}^\circ \in X$, $\mathsf{t}_x[\mathsf{A}]\,\mathsf{o}\,\mathsf{t}_y[\mathsf{A}]$, then, for each order condition $\mathsf{B}^\circ \in Y$, $\mathsf{t}_x[\mathsf{B}]\,\mathsf{o}\,\mathsf{t}_y[\mathsf{B}]$. As an example, consider the salesReps in Table 1.1 having tuple t_3 removed. If salesReps is sorted on column Target, it is also sorted on column Salary. In this case, we can say for example that the pointwise order dependency $\{\mathsf{Target}^>\} \rightsquigarrow \{\mathsf{Salary}^>\}$ holds. For more material on order dependencies, including lexicographical order dependencies, we refer the reader to [58, 60, 59].

Order dependencies generalize functional dependencies, and denial constraints, in turn, generalize order dependencies. Again, we can apply a few transformation rules and rewrite every order dependency into a set of logically equivalent expressions in denial constraint format [59]. For example, consider a pointwise order dependency $X \rightsquigarrow \mathsf{B}$, where every order condition has the same operator o. We can write a denial constraint representation for such order dependency as follows:

$$\forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}\colon \neg \left(\bigwedge_{\mathsf{A} \in X} \mathsf{t}_x.\mathsf{A}\ \mathsf{o}\ \mathsf{t}_y.\mathsf{A} \wedge \mathsf{t}_x.\mathsf{B}\ \bar{\mathsf{o}}\ \mathsf{t}_y.\mathsf{B} \right).$$

Static analysis. The satisfiability problem for order dependencies is studied in [61]. The authors show that satisfaction is independent of the set of columns. In regards to implication, the authors in [60] present a set of inference rules, and the authors in [59] present inference procedures [59]. The inference problem for order dependencies is in coNP-complete [59].

2.2.5 Relaxed dependencies

Relaxed dependencies incorporate extensions in the canonical definition of dependencies so that they can handle certain kinds of errors and exceptions in data. The review study by Caruccio et al. provides a taxonomy of relaxed functional dependencies [23]. For results on the static analysis of relaxed dependencies, we refer the reader to [62, 63] In general, the relaxation concepts for functional dependencies can be extrapolated to other types of dependencies, such as denial constraints.

There are two variants in the relaxation criteria of dependencies; one is relative to the *satisfiability criteria* (or *extent*), and the other is relative to *column comparisons* [23].

Relaxations relative to satisfiability criteria (or extent). Dependencies might be specific to a part of the data. For example, many countries have different policies across different states, so records having different states might obey different rules, hence different dependencies. Besides, some dependencies might not hold entirely in the data due to errors, for example, missing values, typos, or outliers.

Conditional functional dependencies are a well-known example of dependencies that relaxes on the satisfiability criteria. They can express constant patterns that specify those subsets of tuples that satisfy a given functional dependency. Similarly, *conditional denial constraints* such as φ_4 in Table 1.2, can include constants in their predicates to specify subsets of tuples that satisfy a given denial constraint. The subsets of tuples identified by *conditional dependencies* indicate the part of the data that follows particular dependencies.

Another primary example of dependencies that relax on the satisfiability criteria is *approximate dependencies*—sometimes called *partial dependencies*[1]. Here, the satisfiability criteria are relative to the number of violations in a table that does not satisfy a dependency. If that number is below a certain threshold, the approximate functional dependency is valid in that instance. As we discuss later, the *approximation* concept is key in discovering dependencies using possibly inconsistent data. The functional dependencies $\mathsf{Address, City, ST} \rightarrow \mathsf{Zip}$ and $\mathsf{Zip} \rightarrow \mathsf{City, ST}$ are *approximate functional dependencies* if we consider a threshold of one tuple violation, because there is at most one tuple violating each dependency. Similarly, the denial constraints φ_1 and φ_2 in Table 1.2 are *approximate denial constraints*.

An approximate dependency is a dependency satisfied by almost the entire table [26]. Kivinen and Mannila present various measures to define the meaning of "almost". That is, the authors suggest error measures that characterize and quantify dependency errors [62]. Although the work of Kivinen and Mannila focuses on functional dependencies, the proposed error measures can be generalized for other types of dependencies, such as denial constraints.

Relaxations relative to column comparisons The second form of dependency relaxation is relative to the comparison of column values. The idea here is to relax dependencies so that they identify semantic relationships between columns using similarity measures rather than standard relational operators, e.g., equality. The use of similarity measures is useful in production data as they may contain non-uniform representations for the same entity. For example, the same name may occur as "Ben King" or "B. King" in the same column, or related columns. Besides, the similarity concept is useful in handling small variations in numerical domains.

An example of dependency relaxing on column comparison is *metric functional dependencies*, which allow small variations in the dependent (or right-hand side) columns of functional dependencies [64]. This dependency specifies that if two tuples have the same values in columns

[1]We stick to the term approximate dependencies to avoid confusion between partial and conditional dependencies.

X, then their similarity values in columns Y should agree with a specific similarity function on Y, for instance, the edit distance function. In salesReps, the dependency $\mathsf{Address, City, ST} \rightarrow^{\delta=1} \mathsf{Zip}$ is a valid metric functional dependency if we consider the edit distance function and a similarity threshold of $\delta = 1$; observe that tuples t_1 and t_2 are consistent with this dependency. As another example, we refer to a generalization of metric functional dependencies called *differential dependencies*: if two tuples have similar values of X, then their values of Y should also be similar [65]. For example, assuming absolute differences as the distance metric on numerical attributes, a differential dependency in salesReps could be $[\mathsf{Salary}(\leq 1000)] \rightarrow [\mathsf{Bonus}(< 1000)]$. This dependency states that the bonus difference between any two employees within a \$1000 salary difference should be no higher than \$1000.

One approach to relax denial constraints relative to the comparison of column values is to extend the set of built-in operators considered in their predicates. For example, the authors in [54] consider a set of operators $\mathsf{O} = \{=, \neq, <, \leq, >, \geq, \approx\}$— their focus is handling violations of denial constraints. They implement the operator $\approx$ as the edit distance between two strings A predicate is true if this distance is above a certain threshold. A similar approach is considered in a data cleaning system that uses denial constraints, HOLOCLEAN [34]. In this work, we only consider denial constraints relaxing on the extend.

2.2.6 Other types of dependencies

Some data entities might have a variety of syntactically different representations. *Ontology functional dependencies* introduce levels of abstractions in functional dependencies to capture these differences [66]. In a few words, they use synonym and hierarchy relationships to represent and validate the dependencies. A synonym ontology functional dependency, $X \rightarrow_{syn} Y$, states that for each set of tuples with equal values in X, should exist a domain specification for which the values in Y of that set of tuples are synonyms (e.g., "UFPR" is synonymous with "Universidade Federal do Paraná"). On the other hand, an inheritance ontology functional dependency, $X \rightarrow_{inh} Y$, states that for each set of tuples with equal values in X, should exist a domain specification for which the values in Y of that set of tuples are descendants of a least common ancestor (e.g., both "lions" and "tigers" are cats).

Master data management provides methods to define and manage trustful views of data [67, 68]. It is possible to use master data in the definition of *dynamic dependencies* [69, 70]. This type of dependencies determine not only which column values are incorrect, but how to fix them. For example, *editing rules* are defined using relation schemes R and R_m, where the latter is a relation referencing master data [69]. An editing rule states that if there exists a tuple t in relation instance r and a tuple t_m in relation instance r_m such that $t[X] = t_m[X]$, then t should be updated on columns Y using $t[Y] := t_m[Y]$. *Fixing rules* is a another example of dynamic dependency. They combine evidence patterns and negative patterns to expose errors, and express facts to fix them [70]. For example, a fixing rule can combine an evidence pattern "Brasil" for a column Country with a negative pattern {"Rio de Janeiro", "São Paulo"} for a column Capital to

identify common mistakes on pair of columns Country, Capital. If a tuple matches these patterns, then the rule uses the fact "Brasília" to correct the error in Capital. Fixing rules help in repairing ambiguous errors, for example, ("Brasil", "Buenos Aires"). This error could be fixed either as ("Brasil", "Brasília") or as ("Argentina", "Buenos Aires"). The main drawback of dynamic dependencies is the high cost to maintain a solid point of reference for master data.

There has been an increase in interest in revisiting the dependency theory to improve the handling and interpretation of incomplete information in databases. For a comprehensive study on the subject, we refer the reader to [71]—this line of research is orthogonal to the one in this thesis. As a brief example, we mention *embedded functional dependencies*, an extension for functional dependency, which aims at integrating data completeness requirements with the standard requirements of dependencies. Given that $X, Y \subseteq E$, an embedded functional dependency $E : X \rightarrow Y$ extends the notion of functional dependencies because it defines a subset of tuples $\mathsf{r}^E \subseteq \mathsf{r}$ having no missing data (e.g., null values) in the columns in E [72]. The authors study how to apply embedded functional dependencies to identify redundant data values under different interpretations of missing information. Besides, they study the problem of implication in their context and present an inference system for embedded functional dependencies. In [73], the authors study *embedded uniqueness constraints*; their motivation is similar to [72].

2.3 DISCOVERY OF DEPENDENCIES

Designing dependencies by hand can be burdensome, and it is likely to fail if we consider the dynamic changes in data and applications. A compelling alternative is the automatic discovery of dependencies, a problem that is commonly classified as a *data profiling* problem. In a nutshell, data profiling is a set of complex tasks that helps in discovering relevant *metadata* for datasets [22]. Typical examples of metadata include basic statistics (e.g., value distributions), patterns of data values, and dependencies. Data profiling is connected, at least indirectly, to many data management tasks. Thus, it is natural that there has been an extensive number of work addressing data profiling issues. A comprehensive presentation of data profiling tasks and a review of primary contributions on the topic can be found in the survey [22], and the book [74]—a great deal of this material is dedicated to the discovery of dependencies. Besides, a study on the discovery of many types of dependencies can be found in [21].

The problem of dependency discovery is to detect the set of dependencies—expressed in the desired dependency language—that hold on a given relation instance. The number of dependency candidates, that is, dependencies that might potentially hold, in each relation instance is exponential in the number of columns in the relation schema—or even worst depending on the dependency type. Thus, to achieve satisfactory performance, the different dependency discovery solutions employ a variety of different approaches to enumerate and validate dependency candidates. These approaches vary in performance depending on the characteristics of the datasets.

As mentioned earlier, several types of dependencies have been repurposed to handle data inconsistencies. As a result, many discovery algorithms have been developed for these types of dependencies. In the following, we discuss some of the main algorithmic issues and solutions to the dependency discovery problem.

2.3.1 Discovery of functional dependencies, uniques and order dependencies

The approaches for functional dependency discovery can be generally classified into *column-based* or *row-based* approaches [75, 27].

Column-based approaches combine lattice traversals of column combinations and intensive pruning strategies. The intuition in these approaches is that supersets of the functional dependencies, or subsets of non-functional dependencies, discovered previously do not require validation, i.e., they can be pruned. These approaches are known to perform well regarding the size of the relation instance. However, they are sensitive to the size of the schema (i.e., number of columns), so they might provide poor performance for datasets having many columns [27]. Examples of algorithms based on column approaches are TANE [26] and FUN [76].

On the other hand, row-based approaches use cross-comparisons of tuples to find sets of columns sharing the same values. Then, the complement of these sets can be manipulated to produce the set of functional dependencies satisfied by the relation instance. Row-based approaches usually perform better with an increasing number of columns than top-down approaches. However, they perform worst with an increasing number of tuples because of the large number of pair-wise comparisons. Examples of row-based approaches are DEP-MINER[77] and FASTFD [78] algorithms.

A hybrid of column-based and row-based approaches have been proposed as a solution that better scales with increasing numbers of tuples and columns [29]. The HYFD algorithm combines row-based and column-based optimizations, such as sampling and compression, which enable HYFD to scale for larger datasets.

Some of the approaches or techniques in the discovery of functional dependencies are commonplace in dependency discovery; thus, they can be used similarly in the discovery of other types of dependencies.

The discovery of unique column combinations has also been extensively studied [79, 80, 81, 75]. Giannela and Wyss study the problem under the perspective of the Apriori approach for frequent itemset mining [79]. The authors investigate bottom-up, top-down, and hybrid approaches to traverse the powerset lattice of columns. Their bottom-up approach was improved with additional pruning strategies in [80]. A different approach can be seen in GORDIAN algorithm, an example of a row-based approach for unique column combination discovery [81]. The algorithm compresses the dataset into an in-memory prefix tree representation; performs a depth-first traversal of the prefix tree to find collections of non-keys, and finally, complement this collection to produce the set of uniques. A yet alternative approach is modeling the discovery of uniques as a graph processing problem, as in [75]. Finally, a hybrid approach is presented in the

form of the HyUCC algorithm [82]. The algorithm operates in a very similar way that of the HyFD algorithm for functional dependency discovery, so its main advantage is to scale well with both the number of columns and records.

The first algorithm for the discovery of order dependencies, Order, came out only a few years back [83]. Order is based on traversals of a lattice representing order dependency candidates—the traversal approach is somewhat similar to that in Tane algorithm for functional dependency discovery [26]. Each candidate is a list of columns, so the lattice contains all possible lists of columns and; thus, the algorithm incurs a factorial time in the number of columns. The authors in [84] show that Order might prune potentially valid order dependencies from the search space, which leads to incomplete results. Besides, they show that it is possible to map a list-based order dependency into set-based order dependencies in polynomial time and, therefore, it is also possible to design algorithms having exponential worst-case complexity. Their algorithm, FASTOD, is also based on lattice traversals and generally performs better than Order.

The OCDDiscover algorithm considers that each order dependency can be divided into a functional dependency and an order compatibility dependency: a property on two lists of columns that order each other [85]. The search strategy of the algorithm is a breadth-first search, that can run in parallel, where short order dependencies are discovered first. The authors in [86] show that some assumptions in [85] are incorrect, and that OCDDiscover might produce incomplete results. Recently, a hybrid approach called FindUOD has been considered [87], which is deeply inspired by Hydra algorithm (discussed later) for denial constraint discovery. It uses data sampling and correction of preliminary order dependencies that were discovered using the sample to produce the final results.

2.3.2 Discovery of denial constraints

There are two critical limitations concerning the algorithms described in Section 2.3.1. We would need to execute several different algorithms to discover the several different types of dependencies that a dataset might hold. Because the results of each execution is logically independent of each other, we would still need to devise methods to process these results and merge them into a single set of logically valid dependencies. The second limitation regards those dependencies that unique column combinations, functional dependencies, or order dependencies cannot express. We might miss meaningful dependencies, such as complex business rules. The discovery of denial constraint is a natural solution for these two limitations. The results from a single algorithm for the discovery of denial constraints subsume the results from multiple algorithms for the discovery of other types of dependencies. Besides, these results might include many more dependencies due to the higher expressive power of denial constraints.

The problem of denial constraint discovery is to detect all minimal denial constraints that a given a relation instance holds. Denial constraints have a high expressive power because they can express a variety of predicates. However, this very same fact results in a huge search

space for their discovery. The problem is even more challenging than the discovery of other types of dependencies. For example, the number of functional dependencies that potentially hold in a relation instance r with schema R and n columns is $2^n \cdot \left(\frac{n}{2}\right)^2$ [21]. On the other hand, the number of denial constraints that potentially hold in r is $2^{|\mathsf{P}|}$, where P is what we call the predicate space of R [32]. The number of predicates in the predicate space, $|\mathsf{P}|$, is a function of the number n of columns. Although we can restrict some predicate types in P without losing much expressive power, the number of predicates is still large. We can use any pair of columns with any of operator in O. Assuming we only express predicates using a quantifier t_x, t_y and operators in $\mathsf{O} = \{=, \neq, <, \leq, >, \geq\}$, we already have $6 \cdot n \cdot (n-1)$ predicates in P.

All the available algorithms for the discovery of denial constraints follow a similar principle. First, they compare the tuples in the dataset using a variety of mechanisms to compute an *evidence set*. This structure provides enough information to guide the search for denial constraints and to validate denial constraint candidates. An essential question is how to compute evidence sets efficiently, and how to perform the denial constraint search from the evidence set. We postpone the discussion on these questions until Chapter 3, where we present our algorithm and discuss the related work on the discovery of exact and relaxed denial constraints.

2.3.3 Discovery of relaxed dependencies

Generally speaking, discovering relaxed dependencies is harder than discovering their traditional (non-relaxed) counterparts.

Fan et al. present three discovery algorithms for discovering conditional functional dependencies [25]. The first one, CFDMINER, leverages itemset mining techniques (as in [88]) to discover conditional functional dependencies that have only constant patterns. The other two algorithms, CTANE and FASTCFD, focus on general conditional functional dependencies. As their names give away, they are extensions of TANE and FASTFD algorithms. The scalability of CTANE and FASTCFD follows closely their non-conditional counterparts. CTANE scales well in the number of tuples, but it scales poorly in the number of columns of the relation. FASTCFD scales better than CTANE with the number of columns in the relation but requires additional optimizations to better scale with the number of tuples in the relation instance. The problem of discovering conditional functional dependencies has also been studied under the perspective of association rules mining [89].

Most solutions for the discovery of approximate dependencies are adaptations of solutions for the discovery of exact dependencies. For example, TANE algorithm can be modified to discover approximate functional dependencies [26]. The difference between the exact and approximate versions of the algorithm lies in how they validate a candidate dependency: the former validates candidates containing no error; the latter may validate candidates containing errors, given that the error is below a certain threshold. TANE uses data structures called *stripped partitions* (also known as position list indexes) to validate candidates. Such structures have useful properties that enable TANE to estimate the number of tuples that do not satisfy a candidate

(error) efficiently. Another adaptation for the discovery of approximate functional dependencies using striped partitions is described in [90].

Some pruning strategies that work in the discovery of the exact dependency scenario do not work for the discovery of approximate dependencies. For example, approaches such as the hybrid algorithms HyFD, HyUCC, FindUOD, and Hydra aggressively prune the search space. They can discard dependency candidates as soon as they find any single violation for them. Besides, these discarded candidates can be further used to prune other parts of the search space. This principle helps to save a lot of computations; however, it cannot be applied in the discovery of approximate dependencies. This former type of discovery considers the dependency candidate together with the estimation of the dependency error.

Pyro algorithm is currently one of the fastest solutions for the discovery of unique column combinations and functional dependencies [91]. It uses a sampling-based approach to detect promising approximate dependency candidates, which enable the algorithm to prune considerable parts of the search space. Besides, the algorithm proposes the use of a cache system to retrieve some of the position list indexes used to validate the dependency candidates quickly. These two techniques combined result in great performance advantage compared to other algorithms for the discovery of uniques and functional dependencies.

Most of the order dependency discovery algorithms we described have not considered relaxed order dependencies. The authors in [92] give a brief outline of how it would be possible to adapt FASTOD algorithm for the approximate discovery problem, but provide no further evaluation.

Song and Chen present a method for discovering differential dependencies[65]. Their method uses the proportion of tuples matching similarities criteria to estimate support and confidence measures that guide the candidate generation. The authors also describe pruning strategies and an approximated version of their algorithm. Song et al. study the problem of determining distance thresholds for differential dependencies [93]. The idea is to find distance thresholds that maximize the support and confidence of the dependencies with regards to the data. Kwashie et al. present solutions for the discovery of differential dependencies that are based on association rules techniques [94]. They use a measure of interestingness for the candidate dependencies, which helps to reduce the search space.

2.3.4 Discovery of other types of dependencies.

An algorithm for the discovery of synonym and inheritance ontology functional dependencies has been described in [66]. FastOFD algorithm works for the discovery of both exact and approximate dependencies, and it uses an Apriori-like approach [95] to traverse a lattice of attribute sets until all dependencies are discovered. Besides, the authors present a set of inference rules that help to prune the search space.

Diallo et al. present a solution for discovering editing rules from sample and master data [96]. Their method first discovers attribute mappings between sample and master relations

using an approximate discovery of inclusion dependencies (a type of dependency between two tables). Then, it discovers traditional conditional functional dependencies from master data that propagate to sample data through the discovered mappings.

The discovery of embedded functional dependencies is studied in [97]. A naive approach would combine the results from running an algorithm for the discovery of traditional functional dependencies for each subset of tuples of the dataset that satisfy the completeness requirement of embedded functional dependencies. However, such an approach would result in a search space that is much larger than the (already large) search space for traditional functional dependencies. The alternative, and more efficient solution, uses a tree-based data structure to store many correct embedded functional dependencies in each path of the tree [97]. The traversal is inspired by the hybrid approaches for the discovery of traditional dependency. Also, the authors use an inference system to reduce the costs with implied dependencies. Following similar lines of [97], hybrid algorithms for the discovery of embedded unique column combinations is studied in [98].

2.3.5 Dependency ranking

The number of dependencies discovered in a dataset radically increases as the number of columns in the dataset goes up. Even if we rely on static analysis to discard redundant and trivial dependencies, the size of data profiles remains large. Unfortunately, a considerable portion of the discovered profiles is merely spurious or accidental. That is, they are not relevant to the application domain. A dependency is relevant if it can reliably support well-defined applications. For example, a dependency is relevant if it can guide database design, foster data cleaning, or improve query performance.

Database designers can judiciously inspect the relevance of the discovered dependencies and select those dependencies that are pertinent to their target tasks. Such an inspection is likely to be extensive and burdensome; thus, dependency ranking may simplify the whole process. There are various criteria to rank a set of dependencies; we outline some common ones in the following.

Chu et al. propose ranking denial constraints using the weighted average of their *succinctness* and *coverage* [32]. The succinctness concerns the number of distinct symbols (columns and operators) in the predicates of the constraint, and the coverage regards data support based on the proportion of pair of tuples that satisfy subsets of predicates of the constraint. We study these measures in more detail in Chapter 3. A coverage measure is also proposed to rank order dependencies, which is somewhat similar to the one used in denial constraints [84].

Piatetsky-Shapiro and Matheus study a probabilistic generalization of functional dependencies called *probabilistic dependencies* [99]. Their probabilistic analysis and formulas can serve as a measure of the statistical significance of dependencies between two column sets, such as functional dependencies. Sánchez et al. study approximate dependencies under the perspective of association rules [100]. The authors present a correspondence between dependencies and

associations rules and show how to derive support and confidence measures to assess the quality of dependencies.

Ranking dependencies have been used to guide database design. For example, measures based on the redundancy identified in a relation instance can be used to score dependencies [101, 28]. The intuition is that higher-ranked dependencies should produce better schema designs than low-ranked ones. In [102], the authors rank potential primary keys based on their number of columns, the length of column values in their columns, and the position of their columns within the schema.

2.4 DEPENDENCIES IN DATA QUALITY

There has been an increase in concern with low-quality data in decision-making in recent years. This fact has strongly driven research on dependencies, as they are fundamental in data consistency, which in turn, is a key dimension in data quality [6, 7, 103]. Fan gives an overview of dependencies from the perspective of data quality in [48]. A more recent and extensive discussion on the same perspective is presented in [104].

2.4.1 Violations of dependencies

As we discussed in Chapter 1, database management systems may not be able to guarantee database consistency for many types of dependencies natively. Thus, databases might eventually become inconsistent. While some users may not even require automatic fixing of the inconsistencies, they would probably want to know what and where the inconsistencies are, so they could work on data fixes or take those errors into account during decision-making.

Consider a dependency φ and a relation instance r. The violations of the dependency φ is the subset of column values (also called database cells) that cannot coexist in r for φ to hold [104]. We can also refer to dependency violations as the problematic tuples or problematic combination of tuples, rather than the problematic database cells— we use this latter assumption in Chapter 5. In this case, the problem of dependency violation detection becomes finding the tuples (or combinations of tuples) having values that do not agree with the semantics of φ.

The underlying violation detection mechanism of several data cleaning tools is a traditional database management system [35, 105, 34]. The database might underperform in different scenarios, for example, for denial constraints containing complex range predicates. The data cleaning tools inherit the performance issues of database management systems. Besides, their evaluation experiments used small datasets or only simple dependencies, such as functional dependencies. Implementing a dedicated violation module is an alternative. For instance, Chu et al. implement a denial constraint violation detection module based on pairwise comparisons [54]. However, their experimental evaluation also used only a small number of records (i.e., up to $100K$ tuples).

Fan et al. develop a series of SQL-based techniques for detecting violations of conditional functional dependencies [24]. In their method, checking a single conditional functional dependency consists of executing two SQL queries against two tables: the relation instance table and a table containing the *pattern tableau* of constants and the variable fields of the dependency. The first query is a join between the two tables, and it returns the single-tuple violations that do not follow the specification in the pattern tableau. The second query is also a join between these two tables that uses a group by clause to identify the set of tuples that, despite matching the pattern tableau in the left-hand side of the dependency, they fail to match the variable portion in the right-hand side the dependency. The authors extend their techniques to check multiple conditional functional dependencies at a time and evaluate their methods using a commercial database management system.

The issue of scalability in data cleaning is studied by Khayyat et al. [106]. The authors introduce a framework to perform violation detection and database repairing in distributed settings. The core idea is to translate data cleaning rules (expressed in UDF-based form) into jobs that are executed on top of parallel data processing frameworks. Although the approach we describe in Chapter 5 focuses on centralized environments, it is able to detect violations for very large datasets efficiently. Nonetheless, extending our approach for distributed data processing environments is an exciting topic for future work.

HYDRA algorithm, for the denial constraint discovery, contains a specialized violation detection component [55]. Efficient detection of denial constraint violations is critical for the algorithm, so the authors have proposed novel techniques to handle the problem. There are two main ideas in this component: The use of specialized data structures; and the customization of algorithms for different predicate types. We give further details on this component in Chapter 5.

2.4.2 Repairing violations of dependencies

Given an inconsistent database $\mathscr{D}$ and a set Σ of dependencies, how to obtain data consistent with the set Σ from database $\mathscr{D}$? The seminal work of Arenas et al. has introduced two concepts that help to answer such a question: *consistent query answering* and *data repairing* [37]. Both concepts are based on database *repairs*.

Consider a database $\mathscr{D}'$ with the same schema of $\mathscr{D}$, and a function $cost(\mathscr{D}, \mathscr{D}')$ that measures the cost to transform $\mathscr{D}$ into $\mathscr{D}'$ using database inserts, deletes, and updates. A database $\mathscr{D}'$ is a repair of $\mathscr{D}$ if it ensures that $cost(\mathscr{D}, \mathscr{D}')$ is minimal among the possible instances $\mathscr{D}'$ that satisfies the set of dependencies Σ [107]. In other words, the difference between $\mathscr{D}$ and $\mathscr{D}'$ is minimal among all possible $\mathscr{D}'$. The definition of minimality depends on the adopted repairing model; there are a variety of them [104].

Consistent (clean) and inconsistent (dirty) data can coexist in the database in the context of consistent query answering. Whenever applications submit queries to the database, the goal of a consistent query answering approach is to retrieve consistent data at the query answering stages. The consistent data and thus, consistent answers, derive from the space of possible repairs of

the database. Many approximation approaches have been studied to cope with the complexity results in consistent answer processes. We refer to [108, 109, 110, 111] for recent views on the complexity and implementations issues of consistent query answering; and we refer to Bertossi for a comprehensive survey on the subject [107].

Data cleaning based on data repairing seeks to correct the violations of the set Σ of dependencies in the database $\mathscr{D}$ by computing another database $\mathscr{D}'$ that is consistent with Σ and minimally differs from the database $\mathscr{D}$ [1]. In other words, it finds a minimal repair for $\mathscr{D}$. The problem is naturally related to consistent query answering, and it is also quite challenging. The recent advances in the field have helped the development of automatic data repairing tools, for example, [36, 34].

The number of possible repairs for an inconsistent database is exponential; hence the challenge is also to efficiently find good repair candidates [112]. We describe briefly four repair models commonly found in the literature [37, 112]. These definitions restrict the repairing to only column value modifications, also referred to as cell modifications.

A repair $\mathscr{D}'$ is a *cardinality-minimal* repair if it ensures that there exists no repair $\mathscr{D}''$ of $\mathscr{D}$ with less modified cells than $\mathscr{D}'$, where $\mathscr{D}''$ refers to all repairs of $\mathscr{D}$ [113]. A repair $\mathscr{D}'$ is a *cost-minimal* repair if it ensures that there exists no repair $\mathscr{D}''$ of $\mathscr{D}$ such that $cost(\mathscr{D}, \mathscr{D}'') < cost(\mathscr{D}, \mathscr{D}')$ [113]. Let $\mathscr{C}$ denote the subset of the modified cells in a repair $\mathscr{D}'$. The repair $\mathscr{D}'$ is considered a *set-minimal* repair if no subset $\mathscr{C}$ can be converted to its original value in $\mathscr{D}$ without violating any dependency in Σ. The authors of [112] introduce the notion of *cardinality-set-minimal* repair. Similar to set-minimal repairs, a cardinality-set-minimal repair is a repair $\mathscr{D}'$ of $\mathscr{D}$ for which there exist no subset $\mathscr{C}$ that can be transformed back to its original value in $\mathscr{D}$ without violating any dependency in Σ. However, this type of repair allows the remaining cells in $\mathscr{D}'$ to be modified to other values.

Consider a finite set Σ of dependencies, $\mathscr{D}$ and $\mathscr{D}'$ two database instances, and a repair model. The repair checking problem is to decide whether $\mathscr{D}'$ is a repair of $\mathscr{D}$ with regards to Σ and a cost bound [1]. Studies have shown intractability results for this and other related problems involving different dependency languages and repair models [1]. The critical problem is that repairing a given dependency may break others. For example, Bohannon et al. [113] have shown that deciding if there exists a repair $\mathscr{D}'$ of $\mathscr{D}$ is NP-complete for a constant number of functional dependencies, repair models based on value modifications, and a limited number of modifications in $\mathscr{D}'$. Thus, data repairing is highly nontrivial, and repairing algorithms are mostly heuristics [1].

Bohannon et al. present greedy approaches to discover data repairs regarding functional dependencies and inclusion dependencies [113]. Their algorithms employ *equivalence classes*, that is, groups of cells that should have the same value. First, all the database cells are assigned to their respective equivalence classes. The intuition behind the algorithm is to isolate the procedures that choose which cells should have the same value from the procedures that choose which cells should be assigned to the same equivalent set. By doing so, the algorithm mitigates

poor quality local modifications, for example, a name that was misspelled in one place may have its correct version at other views of the domain. The greedy approach keeps merging equivalence classes until all dependencies are satisfied. The algorithm has inspired other extensions, such as an algorithm for repairing data based on conditional functional dependencies [114].

Chu et al. present a data repairing algorithm that repairs violations of different types of dependencies holistically [54]. Denial constraints serve as their data quality dependency language because, as mentioned earlier, denial constraints subsume many other types of dependencies. First, the algorithm builds a *conflict hypergraph* from the database cells and the violations of denial constraints. Each cell in the database becomes a node in the conflict hypergraph, and each violation is encoded as a hyperedge of the conflict hypergraph. The database cells that participate in multiple violations are those that are more likely to contain errors. The algorithm finds a minimum vertex cover for the hypergraph, which represent the problematic cells. Then, it uses an auxiliary data structure called *repair context* to collect the information required to repair the erroneous cells. Two procedures can generate possible repairs according to the content of the repair contexts: value frequency mapping and quadratic programming. The database is clean when the conflict hypergraph is empty or when some termination criteria is met. NADEEF [35] is an open-source data cleaning tool that uses the techniques presented in [54].

Dynamic dependencies (e.g., fixing rules and editing rules) provide means to fix the errors directly. The authors of [69] propose editing rules to repair data based on master data. The solution finds *certain fixes* based on *certain regions* and editing rules. Certain fixes are updates for which is guaranteed to exist the information needed for correcting an erroneous tuple. Certain regions are sets of columns for which users assure their correctness.

HoloClean brings together denial constraints, master data, and statistical analysis of data to form a probabilistic model for data repairing [34]. The main intuition of HoloClean is that the probabilistic model is a natural solution to integrate different signals for a particular task; in this case, data cleaning. The first step in HoloClean is to separate database cells into erroneous or clean cells. The tool can use any error detection solution as long as the output represents identifiers for the erroneous and clean cells of the database. HoloClean assigns a random variable to each cell of the database and then compile a graphical model that describes the distribution of these variables. The tool uses a declarative probabilistic inference framework called DeepDive [115]. It enables the statistical learning and inference of the models. The random variables associated with the clean cells are used as labeled examples for learning the parameters of the model. Finally, the value of the random variables associated with the erroneous cells is inferred using approximate inference.

2.4.3 Repairing dependencies

The solutions we described so far assume that the input set of dependencies is correct. What if this set, or any of its subsets, is wrong? An alternative is to trust the data but update or

discard the erroneous dependencies. The intuition here is that if data are ever-evolving, then the semantics of data might be evolving as well [116].

Intuitively, the semantics of the application domain and data updates may suggest natural dependency evolutions, for example, a conditional functional dependency pattern [Country = 'Brazil'] $\rightarrow$ [FuelTaxes = 30.0] evolving into a new pattern [Country = 'Brazil'] $\rightarrow$ [$fuelTaxes$ = 50.0]. Golab et al. describe an approach to discover the conditional parts, or pattern tableaux, of conditional functional dependencies [117]. The goal is to generate good pattern tableaux by maximizing the number of tuples matching the pattern while minimizing the number of violating tuples. The authors show that the problem of generating such parsimonious tableaux is NP-hard and propose an approximate solution. Their greedy algorithm enumerates all possible tuple patterns from the active domain, and then compute the support (quantity of matching tuples) and confidence (quantity of violations) for these patterns. The algorithm iteratively picks a pattern tableaux, estimate the support for the remaining tuple patterns, and validates the chosen tableau against given thresholds.

Chiang and Miller consider both data and functional dependency repairings [118]. Their principle for dependency repairing is to add columns to the body of the violated dependencies so that these new dependencies become consistent with the data. The authors describe a cost model for repairing data and dependencies that quantifies the trade-off between repairing data errors and evolving constraints. Their cost model is based on the *minimum description length*. Given a database instance $\mathscr{D}$ and a set Σ of functional dependencies such that Σ is inconsistent in $\mathscr{D}$, the goal is to find repairs $\mathscr{D}'$ and Σ' at a minimal cost. Beskales et al. [119] follow a close motivation to [118], and they incorporate the notion of *relative trust* between the two types of repairings. The idea is to limit the number of data changes with a threshold and generate multiple possible repairs for user validation.

Mazuran et al. also present a method to support evolving functional dependencies [120]. The method repairs dependency violations by adding more columns to the left-hand side X of the dependency. It is based on the confidence of a functional dependency $f : X \rightarrow Y$, given by the ratio between the number of distinct values for the set of columns X, and the number of distinct values for the set of columns XY. A dependency f having a confidence value lower than one means that f has been violated and needs repairing. The proposed method first sorts the set of functional dependencies according to the average of two metrics: the degree of inconsistency, which is based on the confidence value, and the conflict score, which is based on the common columns a functional dependency has among the set of all dependencies. The dependency repairing follows the order in this sorting step. Consider A a column candidate to extend the column set X of f. The method computes the confidence of dependencies $f' : X\mathsf{A} \rightarrow Y$ and returns an ordered list of candidate columns sorted in descending order of confidence. In case the dependencies f' produce the same confidence, the method can further estimate a goodness measure (the modular difference between the projection of $X\mathsf{A}$ and Y) to decide which column to choose.

2.5 DEPENDENCIES IN QUERY OPTIMIZATION

The benefits of using dependencies for query optimization have been studied for decades [121]. The correlation detection via sampling (CORDS) recommends sets of attributes for which query optimizers should maintain additional statistics [122]. To do so, CORDS discovers approximate functional dependencies with a sample based approach which refines sets of candidate attribute pairs, chosen from the catalog statistics and the sampled attribute values.

EXORD is a three-phase framework for exploiting attribute correlations in big data query optimization [123]. It considers source-to-target attribute mappings as correlations. The first phase of EXORD is responsible for validating an initial user-defined set of correlations. It works on simple statistics (e.g, the number of records violating a correlation) to only keep correlations that fall under user-defined thresholds. The second phase uses a cost model to select correlations for deployment. The authors look into an interesting optimization problem: how to select a subset of correlations with the objective of maximizing the total benefit (i.e, correlation applicability). The exploitation phase is responsible for rewriting the queries so that they exploit more efficient access plans.

In Chapter 6, we consider semantic query optimizations, particularly, how to use functional dependencies to modify queries so that performance is enhanced but semantics preserved. Some commercial optimizers (e.g, [124]) incorporate rewriting strategies into the planning phases. In [59], the authors investigate the use of order dependencies (a variant of functional dependencies) for order optimization. [125] study variations of join elimination and predicate introduction. Unfortunately, most of the studies on semantic query optimization require the user to specify a set of constraints. In contrast, we present a tool that eliminates this manual interaction by employing automatic discovery and selection of dependencies.

2.6 DEPENDENCIES IN DATABASE DESIGN

Poorly design databases, and in particular, poorly design relation schemas, can lead databases to face information redundancies and update anomalies. Dependencies—primary functional dependencies and some of their variations—serve as a formal mechanism for analysis of relation schemas. They enable us to identify low-quality relation schemas, and they provide means to transform such schemas into better-quality ones.

Most database textbooks cover the fundamentals of dependencies applied in database design, for example, [9, 8, 15]. They describe some well-known *normalization* processes that guide the design of good-quality relation schemas. In a few words, a normalization process relies on a set of dependencies and their implications to identify flaws in a relation schema. Then, it decomposes the flawed schemas into other schemas that meet properties for good relation schema design. *Normal forms* express these properties. For example, Boyce-Codd normal form (or BCNF for short) states that a relation schema R satisfies BCNF if for every functional dependency $X \rightarrow \mathsf{A}$ defined for R, the left-hand side X is a *superkey*—a set of columns that

contains a key [46]. The ultimate goal of normalization processes is to replace a problematic relation schema with other relation schemas that do not lead to redundancies or anomalies.

NORMALIZE is an algorithm for automating the normalization process [102]. It takes a relation instance along with its relation schema as input and produces a set of relation schemas that is compliant with BCNF as output. The algorithm first discovers the set of all minimal functional dependencies holding in the relation instance given as input. Then, it extends the discovered dependencies to maximize their right-hand side; the authors describe efficient algorithms for this step. The maximization helps the algorithm to identify keys and BCNF violations. NORMALIZE then identifies the functional dependencies violating BCNF, rank them, and select the top-scored dependency for normalization (a user might interact with the algorithm in this phase). Finally, the algorithm runs a few strategies to select keys for every (decomposed) relation in the output (users might also be involved).

The normalization process has been rethought into the context of embedded functional dependencies [126]. The framework can capture data redundancies regardless of the different interpretations of missing data. In addition to establishing an inference system, the authors present a generalization of BCNF.

Chapter 3

Discovery of Denial Constraints

Defining dependencies by hand requires judging the structure and content of a database. The task requires expertise and time, and it is error-prone considering how complex and dynamic production datasets can be. As discussed in Section 2.3, the profiling of datasets to discover dependencies has emerged as a promising alternative to the manual design of dependencies [22]. The discovery of denial constraints discovery is particularly helpful for the complex datasets emerging from extracted data, e.g., knowledge graph construction or web tables repositories.

A single denial constraint discovery algorithm can replace the several algorithms required to discover the various types of dependencies a dataset might hold. Besides, because denial constraints have high expressive power, they can capture business rules that could not be expressed by more restrict types of dependencies. Discovering denial constraints helps to capture non-obvious complex business rules. Recent approaches related to data cleaning have used denial constraints as the *de facto* integrity constraint language [34, 127, 128]. The discovered denial constraints naturally can serve as the input of such approaches.

The computational complexity of discovering dependencies regards the number of tuples and columns of a relation [22]. The complexity of discovering denial constraints, in turn, regards additional challenges because each denial constraint is expressed as a set of predicates rather than a set of columns. The denial constraint search space consists of any subset of the predicates drawn for a relation. Each column adds many denial constraint candidates to the search space because each additional column can generate predicates of various types: Equalities, inequalities, and comparisons across columns. Therefore, discovering denial constraints requires efficient techniques to traverse the search space and validate denial constraint candidates.

Discovering approximate denial constraints is even more challenging than discovering exact denial constraints because the former task requires an algorithm to keep track of the number of tuple pairs that violate each candidate. This requirement prohibits the use of aggressive pruning techniques, which uses the fact that a single violation is enough to invalidate a candidate— that is not true for approximate dependencies.

In this chapter, we present a novel algorithm, DCFINDER, to discover both approximate and exact denial constraints efficiently. DCFINDER first iterates over the data to build auxiliary

data structures that summarize column values and tuples containing those values. Then, the algorithm uses these auxiliary structures to build compact representations of tuple pairs and their satisfied predicates. This step uses information on predicate selectivity for performance. With the compact tuple pair representation, DCFINDER can directly generate and validate exact and approximate denial constraint candidates. The output of the algorithm is the set of all minimal denial constraints holding in the input dataset.

The capability of measuring the interestingness of discovered denial constraints is essential since it helps users decide which denial constraints are relevant for their application. The design of DCFINDER enables the algorithm to calculate and output different measures of interestingness for the discovered denial constraints. We can use this additional information to rank the discovered results and provide users with different perspectives on the interestingness of denial constraints.

In summary, our contributions in this chapter are as following:

- We present the novel DCFINDER algorithm for the discovery of approximate and exact denial constraints.
- We provide an experimental comparison of DCFINDER to all previously existing denial constraint discovery algorithms, showing that DCFINDER is the most efficient algorithm for the discovery of approximate denial constraints and, at times, better than state of the art even for the discovery of exact denial constraints.
- We provide a study on different interestingness measures of discovered denial constraints and their efficient calculation to enable denial constraint selection.

The rest of the chapter is organized as follows: In Section 3.1, we discuss previous solutions for denial constraint discovery. In Section 3.2, we present key definitions and notations. In Section 3.3 we present an overview of DCFINDER. We split the description of our algorithm into preprocessing (Section 3.4); evidence set building (Section 3.5); and denial constraint search, followed by denial constraint interestingness (Section 3.6). In Section 3.7 we present our experimental evaluation. Finally, in Section 3.8 we present a summary of this chapter.

3.1 PREVIOUS ALGORITHMS FOR DENIAL CONSTRAINT DISCOVERY

Discovery of exact denial constraints. FASTDC was the first algorithm for denial constraint discovery [32]. By the time we started this research project, it was the only algorithm on the subject. FASTDC compares every tuple pair of the input dataset to compute *evidence*, that is, what are the predicates each tuple pair satisfies. The result of this computation is called *evidence set*. The authors of FASTDC have shown how to transform the problem of denial constraint discovery into the problem of discovering *covers* for an evidence set. FASTDC uses the predicate distribution in the evidence set to guide a depth-first search and discover these covers. The

approach based on the evidence sets in FASTDC has inspired all the other algorithms for the discovery of denial constraints.

The reason why approaches based on evidence set are suitable for denial constraint discovery is that they scale relatively well in the number of columns of datasets. An alternative approach would be based on lattice traversals, which would arrange all possible denial constraint candidates in a lattice of column combinations and then use the data instance to validate the candidates, similar to what some functional dependency discovery algorithms do [26, 129]. Extensive experimental evaluation has shown how lattice-based algorithms, like [26, 129], quickly run into memory or performance issues for datasets with a relatively large number of columns [27]. The search space is even larger for denial constraint discovery than it is for functional dependency discovery because a single column may add many predicates into the search space. Thus, building lattices of predicate combinations might be prohibitive.

Instead of building huge lattices, the algorithms for the discovery of denial constraints follow the evidence set approach proposed in FASTDC [55, 130, 41]. Evidence sets are comparable to the difference-sets used in the discovery of functional dependencies [77, 78]. These structures help us to define the search space based on instance observations rather than exhaustive candidate enumeration. As observed in [27], the algorithms based on difference-sets can keep reasonable memory footprints in generating and validating candidates. With this in mind, building evidence sets efficiently plays a significant role in denial constraint discovery.

During the building of evidence sets, FASTDC algorithm suffers from performance issues due to the quadratic computation in the number of tuples. This fact drove us to design a faster algorithm called BFASTDC [130]. BFASTDC improves the building of evidence sets based on two key principles. It combines tuple identifiers from related column values and avoids testing every pair of tuples for every predicate. Besides, it exploits the implication relation between predicates to operate at a bit level.

Despite the considerable performance improvement over FASTDC, BFASTDC algorithm still requires many logical operations to calculate which predicates are satisfied by tuple pairs, which hinders performance. This fact led us to design a second algorithm, which is described in this chapter. DCFINDER also uses column value indexing to avoid the expensive tuple pair comparison of FASTDC. To drive efficiency even further, it uses predicate selectivity to avoid the unnecessarily large number of logical operations required by BFASTDC. In this thesis, we describe only DCFINDER in detail for the following reasons. The key insights of BFASTDC are also present in DCFINDER; thus, the description of DCFINDER also enlightens the central aspects of BFASTDC. Besides, the experimental evaluation of DCFINDER is more exhaustive, since it includes all algorithms for the discovery of denial constraints that were available previously to its proposal. The full description of BFASTDC can be found in [130].

Other algorithmic insights for the discovery of denial constraints can be seen in the HYDRA algorithm [55]. It employs sampling of tuple pairs in order to save a considerable amount of time when calculating the evidence set. From the sample, HYDRA builds an intermediary

evidence set and derives an intermediary set of denial constraints. Then, from this set of constraints, the algorithm corrects the tuple pair sample and determines the complete evidence set. In an approach comparable to FASTDC, the algorithm extracts the final denial constraints from the complete evidence set.

Discovery of relaxed denial constraints. Having error-free data to derive denial constraints is unrealistic, so it is reasonable to relax their satisfiability criteria. The discovery of conditional dependencies uses the values in the domain of columns (called constants, for short) to specify the parts of the data a dependency holds. In the case of the discovery of conditional denial constraints, this specification is through predicates involving constants, i.e., predicates of the form $t_x.A_i$ o c for constants c in $dom(A_i)$. The authors of FASTDC present a modification to their algorithm that can discover conditional denial constraints, called C-FASTDC.

The number of constants can be quite large, hence, a large multiplication in the number of possible predicates to form conditional denial constraints. The complexity of denial constraint discovery is greatly affected by the number of predicates, so it becomes infeasible for a discovery approach to consider extensive sets of predicates. The main idea of C-FASTDC is to filter out those predicates involving constants which are not frequent, or in other words, predicates having a low *support*. A predicate has high support if the number of tuples that satisfy it is above a given threshold. C-FASTDC uses an an Apriori approach [131] to search for high-support sets of predicates. For each set of predicates and the subset of tuples satisfying its predicates, C-FASTDC calls the regular FASTDC algorithm to discover non-conditional denial constraints holding in that subset. The result is a combination between the high-support predicates and the non-conditional denial constraints discovered.

We can substitute the call to FASTDC in C-FASTDC algorithm by a call to any other denial constraint discover algorithm. BFASTDC implements this conditional denial constraint discovery approach [130]. The improvements in runtime come from the discovery of the non-conditional parts of denial constraints. However, BFASTDC presents no further techniques or optimizations for discovering conditional denial constraints, so we do not include this results in this thesis.

In our study, we consider the possibility that a few tuple pairs may not satisfy a valid denial constraint due to imperfect data. Still, the discovery algorithm should be able to find that valid (but approximate) denial constraint. It turns out that discovering approximate denial constraints is even more challenging than discovering exact denial constraints. For every approximate denial constraint discovered, the algorithm must guarantee that the number of violations for that denial constraint is no greater than a given threshold. To do so, it needs to know how many tuple pairs may still violate a candidate denial constraint. It is possible to obtain this information from the evidence sets, as long as the algorithm keeps information on evidence set multiplicity. FASTDC, BFASTDC and DCFINDER can integrate a few modifications in their operation to provide such information and discover approximate denial constraints.

HYDRA algorithm, however, works under different assumptions; and it is yet to be shown how the algorithm can be adapted to discover approximate denial constraints.

HYDRA algorithm assumes that a denial constraint is valid if there does not exist one single tuple pair violating that denial constraint. Such an assumption does not hold for approximate denial constraints. HYDRA leaps over the evidence search space to save computations on duplicate pieces of evidence. The technique may reduce computation time, but loses the evidence set multiplicity. We observed that the number of evidence produced by HYDRA is only a fraction of the evidence required to discover approximate denial constraints (more details in Section 3.7). An adaptation of HYDRA algorithm to discover approximate denial constraints would require significant changes in the algorithm, which is beyond the scope of this thesis. In our experiments, however, we use the algorithm as a baseline to evaluate how DCFINDER compares to a specialized exact denial constraint discovery solution.

3.2 BACKGROUND

Let us walk through the semantics of the *employees* relation in Table 3.1, which we use as the running example in this chapter. Mind that we now use new identifiers for each new denial constraint. Any two employees that have the same $\{\mathsf{Name}, \mathsf{Phone}\}$ values have the same $\{\mathsf{Position}\}$ value. This statement is a functional dependency, which is translated into a denial constraint as follows: If a tuple pair t_x, t_y of *employees* satisfies the predicates $t_x.\mathsf{Name} = t_y.\mathsf{Name}$ and $t_x.\mathsf{Phone} = t_y.\mathsf{Phone}$, it cannot satisfy the predicate $t_x.\mathsf{Position} \neq t_y.\mathsf{Position}$. The following denial constraint expresses this dependency:

$$\varphi_1 : \neg(t_x.\mathsf{Name} = t_y.\mathsf{Name} \wedge t_x.\mathsf{Phone} = t_y.\mathsf{Phone} \wedge t_x.\mathsf{Position} \neq t_y.\mathsf{Position})$$

Table 3.1: An instance of the relation *employees*.

	Name	Phone	Position	Salary	Hired
t_0	W. Jones	202-222	Developer	$2.000	2012
t_1	B. Jones	202-222	Developer	$3.000	2010
t_2	J. Miller	202-333	Developer	$4.000	2010
t_3	D. Miller	202-333	DBA	$8.000	2010
t_4	W. Jones	202-555	DBA	$7.000	2010
t_5	W. Jones	202-222	Developer	$1.000	2012

The relationship between the columns Position, Salary and Hired shows that for any two employees with the same position, the longer-standing employee always earns the highest salary. If a tuple pair t_x, t_y of *employees* has the same position, then the predicate $t_x.\mathsf{Position} =$

t_y.Position is true. If that is the case and t_x.Hired $<$ t_y.Hired is true, then t_x.Salary $<$ t_y.Salary is false. This business rule is expressed as a denial constraint as follows:

$$\varphi_2\colon \neg(t_x.\text{Position} = t_y.\text{Position} \wedge t_x.\text{Hired} < t_y.\text{Hired} \wedge t_x.\text{Salary} < t_y.\text{Salary})$$

The denial constraints in the previous examples are fully satisfied by the data in Table 3.1. Recall that a denial constraints with this feature is usually called exact denial constraints. In ideal settings, data is error-free, and the constraints are fully satisfied. In reality, data all too often present inconsistencies. The root cause of inconsistencies vary greatly, for instance, from schema evolution to erroneous data imputation not caught by the (un)defined constraints.

One of the workarounds for potential data errors is to relax the constraints so that they admit a certain degree of inconsistency, but still hold for most of the data [23]. Denial constraints with this relaxation feature are called relaxed or, here, approximate denial constraints. In the *employees* relation, we can see that there are two (non-reflexive) tuple pairs that satisfy the predicates t_x.Name $=$ t_y.Name and t_x.Phone $=$ t_y.Phone simultaneously, t_0, t_5 and t_5, t_0. Those two predicates define an approximate denial constraint, which reads: there cannot exist any two employees with the same values of $\{\text{Name}, \text{Phone}\}$. This constraint seems a reasonable key candidate for the *employees* instance and reveals the potential inconsistency between tuples t_0 and t_5 as duplicates. This dependency is expressed as an approximate denial constraint as follows:

$$\varphi_3\colon \neg(t_x.\text{Name} = t_y.\text{Name} \wedge t_x.\text{Phone} = t_y.\text{Phone})$$

The above example shows how meaningful denial constraints may be "hidden" amid inconsistent data. In this work, we are also interested in relaxing the denial constraint satisfiability constraint so that if a denial constraint has just a small number of violations, it still can be considered valid. An approximate denial constraint allows a limited number of violations to exist in a table r before it is considered invalid in r.

We follow related work and use the proportion between the number of violating tuple pairs and the total number of tuple pairs in a table as a denial constraint error measure [32, 91]. This measure quantifies the degree of approximation of a denial constraint φ in r, and it is calculated as follows [62]:

$$g_1(\varphi, r) = \frac{\left|\left\{(t_x, t_y) \in r \mid (t_x, t_y) \not\models \varphi\right\}\right|}{|r| \cdot (|r| - 1)}$$

We use the degree of approximation above to relax the satisfiability criteria of denial constraints, and define approximate denial constraints in the following.

Definition 3 (Approximate Denial Constraint)**.** Given an error threshold ε, $0 \leq \varepsilon < 1$, a denial constraint φ is ε-approximate in r if and only if its degree of approximation $g_1(\varphi, r)$ is below ε.

Evidence set. Let e_{t_x,t_y} be the set of predicates that tuple pair t_x, t_y satisfies, that is, $e_{t_x,t_y} = \{p \mid p \in P, t_x, t_y \models p\}$. We refer to these subsets as tuple pairs evidence e (or simply evidence e when the context is clear) [32]. Given a relation instance r and a predicate space P, the evidence set E_r is the set of evidence w.r.t. r and P, that is, $E_r = \{e_{t_x,t_y} \mid \forall t_x, t_y \in r\}$. The authors in [32] have shown that it is possible to obtain the set of minimal denial constraints from the evidence set E_r. Besides, the evidence set can be used to efficiently calculate the degree of approximation of each candidate denial constraint.

Problem definition. Given a relation instance r, and an error threshold ε, the problem of approximate denial constraint discovery is to find all ε-approximate minimal denial constraints that hold on r. The discovery of exact denial constraints is a particular case of this problem, where the error threshold is set to zero. Besides, this discovery problem can be viewed as enumerating *minimal covers* (also known as *minimal hitting sets*) for the evidence set.

3.3 OVERVIEW OF DCFINDER

Figure 3.1 depicts the building blocks of our denial constraint discovery algorithm. From the dataset schema, DCFINDER defines a predicate space; and from the dataset records, the algorithm assembles data structures called position list indexes (PLIs). Some types of predicates are most likely to have low selectivity (i.e., when a predicate is satisfied by many tuple pairs). DCFINDER takes this into account to divide the predicate space into likely/unlikely predicate sets. The idea is to presume that a piece of evidence satisfies the least selective predicates. DCFINDER then allocates arrays of evidence where every element holds the set of "most likely satisfied" predicates. The algorithm uses PLIs to compute references to tuple pairs that do satisfy the "unlikely satisfied" predicates. Performing simple logical operations for each of these references brings the arrays of evidence to their consistent state. Finally, the algorithm uses a simple hash table to map the elements of these arrays into the final evidence set.

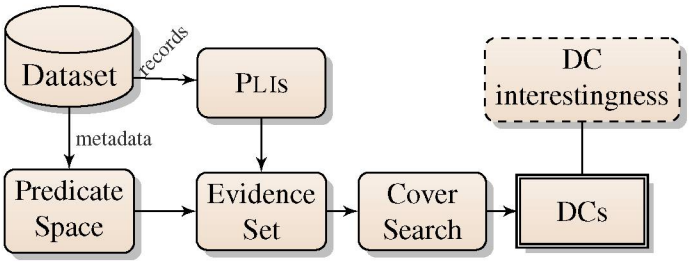


Figure 3.1: Building blocks of DCFINDER .

The evidence set is a compact representation of tuple pairs and their satisfied predicate sets. It enables efficient validation of denial contraint candidates. To discover all minimal denial contraints, DCFINDER uses a depth-first search (DFS) strategy based on evidence set coverage of denial contraint candidates. The last, optional, step is to rank denial contraints based on interestingness measures to help users filter the discovered results.

3.4 DATASET TRANSFORMATION

DCFINDER transforms a relational dataset into a predicate space and PLI index structures, as described next.

3.4.1 From schema into predicate space

Any subset of the predicate space P is a denial constraint candidate, and the denial constraint search space is of size $2^{|\mathsf{P}|}$. We follow related work and apply some restrictions to our predicate space [32, 55]. As showed in [32], restricting the predicate space helps prune meaningless results and reduces computational costs. We distinguish the attribute types whether they are character strings, longs, or doubles, and we use the set of built-in operators $\mathsf{O} = \{=, \neq, <, \leq, >, \geq\}$. For numeric attributes, we define predicates with all operators $\mathsf{o} \in \mathsf{O}$; for non-numeric attributes, we define only predicates with operators $\mathsf{o} \in \{=, \neq\}$. Predicates across two different attributes are regarded only as long as their attributes have the same type and share at least 30% of common values [32]. Figure 3.2 illustrates the predicate space defined for the relation *employees* in Section 3.2.

$\mathsf{p}_1 : \mathsf{t}_x.\mathsf{Name} = \mathsf{t}_y.\mathsf{Name}$	$\mathsf{p}_{10} : \mathsf{t}_x.\mathsf{Salary} \leq \mathsf{t}_y.\mathsf{Salary}$
$\mathsf{p}_2 : \mathsf{t}_x.\mathsf{Name} \neq \mathsf{t}_y.\mathsf{Name}$	$\mathsf{p}_{11} : \mathsf{t}_x.\mathsf{Salary} > \mathsf{t}_y.\mathsf{Salary}$
$\mathsf{p}_3 : \mathsf{t}_x.\mathsf{Phone} = \mathsf{t}_y.\mathsf{Phone}$	$\mathsf{p}_{12} : \mathsf{t}_x.\mathsf{Salary} \geq \mathsf{t}_y.\mathsf{Salary}$
$\mathsf{p}_4 : \mathsf{t}_x.\mathsf{Phone} \neq \mathsf{t}_y.\mathsf{Phone}$	$\mathsf{p}_{13} : \mathsf{t}_x.\mathsf{Hired} = \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_5 : \mathsf{t}_x.\mathsf{Position} = \mathsf{t}_y.\mathsf{Position}$	$\mathsf{p}_{14} : \mathsf{t}_x.\mathsf{Hired} \neq \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_6 : \mathsf{t}_x.\mathsf{Position} \neq \mathsf{t}_y.\mathsf{Position}$	$\mathsf{p}_{15} : \mathsf{t}_x.\mathsf{Hired} < \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_7 : \mathsf{t}_x.\mathsf{Salary} = \mathsf{t}_y.\mathsf{Salary}$	$\mathsf{p}_{16} : \mathsf{t}_x.\mathsf{Hired} \leq \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_8 : \mathsf{t}_x.\mathsf{Salary} \neq \mathsf{t}_y.\mathsf{Salary}$	$\mathsf{p}_{17} : \mathsf{t}_x.\mathsf{Hired} > \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_9 : \mathsf{t}_x.\mathsf{Salary} < \mathsf{t}_y.\mathsf{Salary}$	$\mathsf{p}_{18} : \mathsf{t}_x.\mathsf{Hired} \geq \mathsf{t}_y.\mathsf{Hired}$

Figure 3.2: Predicate space for the *employees* relation.

3.4.2 From tuples into PLIs

PLIs represent the unique values of a dataset [27]. Consider the attribute $\mathsf{A}_i \in \mathsf{R}$. A cluster is an entry $c = \langle k, l \rangle$, where key k is a value from the projection operation $\pi(\mathsf{A}_i)$ and value l is a list of tuple identifiers of the relation instance having the same value k, i.e., $\forall x \in l$ then $t_x[\mathsf{A}_i] = k$. The list l maintains its elements in ascending order. A PLI $\Pi(\mathsf{A}_i)$ is the set of all cluster entries of A_i in r. The numeric PLIs are sorted by the entry keys in descending order. Figure 3.3 shows the PLIs of the *employees* relation.

PLIs are commonly used in attribute dependency discovery, and are also known as *stripped partitions* [26]. In these works, intersecting the values of PLIs helps to validate dependency candidates. In our context, PLIs are used as an intermediate data structure that helps generating evidence sets. With PLIs, we can efficiently answer the question: which tuple pairs

Name

k	l
W. Jones	{0, 4, 5}
B. Jones	{1}
J. Miller	{2}
D. Miller	{3}

Position

k	l
Developer	{0, 1, 2, 5}
DBA	{3, 4}

Salary

k	l
8,000	{3}
7,000	{4}
4,000	{2}
3,000	{1}
2,000	{0}
1,000	{5}

Phone

k	l
202-222	{0, 1, 5}
202-333	{2, 3}
202-555	{4}

Hired

k	l
2012	{0, 5}
2010	{1, 2, 3, 4}

Figure 3.3: Transformation of the records of *employees* into PLIs.

satisfy a given predicate p? DCFINDER simply iterates over cluster combinations to generate these tuple pairs; the details are given in Section 3.5.

Building PLIs takes linear time as it requires only projection operations to collect the distinct attribute values and their associated tuple identifiers. PLIs are used to look clusters up. Non-numeric clusters are stored in hash tables so looking them up takes constant time. Numeric clusters are stored as sorted arrays so that it is possible to look keys up using binary search. The binary search is required for looking up inequalities. For instance, given a key k, we can ask what is the next cluster whose key is greater than k.

3.5 EVIDENCE SET GENERATION

One may think that storing evidence sets requires significant resources, because they represent all tuple pairs. However, different tuple pairs may draw redundant evidence, i.e., they may satisfy the very same set of predicates. As a matter of fact, the number of distinct pieces of evidence was just a fraction of the total number of tuple pairs of the datasets in our experiments. As a result, keeping only the distinct evidence saves a huge amount of space. But the computational costs of materializing tuple pair evidences may still be high. To significantly reduce also these costs, DCFINDER uses attribute indexing and predicate selectivity with a novel approach.

Let us first assume that the pieces of evidence of r are stored into a virtual array B. Each tuple pair is assigned an identifier tpid to index B as in Equation 3.1.

$$\mathsf{tpid}(\mathsf{t}_x, \mathsf{t}_y, \mathsf{r}) = |\mathsf{r}|x + y \tag{3.1}$$

Our goal is to put B into a consistent state. Every element $B[\mathsf{tpid}]$ must hold only the predicates satisfied by tpid. The naive approach would fill each evidence of B by evaluating

every tuple pair for every predicate. This approach performs poorly due to the high number of tuple pair accesses and predicate evaluations. DCFINDER avoids directly comparing every tuple pair by benefiting from two main insights: First, some predicates may have low selectivity, and if so, are satisfied by many tuple pairs. Second, we can efficiently build attribute value associations between tuple pairs and their satisfied predicates using PLIs. DCFINDER is designed based on these two insights to minimize the number of operations within the evidence array B. This drastically reduces the performance penalties from the quadratic tuple pair space, thus helping the efficiency of DCFINDER.

DCFINDER builds evidence sets, in the three stages: Evidence initialization, reconstruction, and counting.

3.5.1 Evidence initialization

DCFINDER initializes an array B so that many of the elements of B are close to their consistent state. Consider an evidence e to be stored in $B[\mathsf{tpid}]$. The probability of a predicate p to occur in e is simply the probability of tpid to satisfy p, i.e., the selectivity of predicate p. Tuple pairs are more likely to satisfy the least selective predicates. Under this assumption, DCFINDER fills in a piece of general evidence $\mathsf{e}_{\mathsf{ahead}}$ with some of the least selective predicates, and then instantiates every element of B as a copy of $\mathsf{e}_{\mathsf{ahead}}$. The chances are high that many elements of B are already consistent for some $\mathsf{e}_{\mathsf{ahead}}$ predicates. For instance, all the tuple pairs of the employees relation satisfy the predicate $\mathsf{t}_x.\mathsf{Salary} \neq \mathsf{t}_y.\mathsf{Salary}$. This form of evidence ahead initialization is what differs DCFINDER from BFASTDC. The latter algorithm initializes the array B with empty elements; as a consequence, it is required to use many more logical operations to fill each evidence correctly.

Recall Figure 2.1 and predicate implication that tells us that each predicate $\mathsf{p}_1 : \mathsf{A}_i \ \mathsf{o} \ \mathsf{A}_j$ implies every predicate $\mathsf{p}_2 : \mathsf{A}_i \ \mathsf{o}' \ \mathsf{A}_j$, where $\mathsf{o}' \in \mathsf{o}_{\Rightarrow}$. Therefore, DCFINDER also includes the implications $impl(\mathsf{p})$ of p into $\mathsf{e}_{\mathsf{ahead}}$, for every p it has included into $\mathsf{e}_{\mathsf{ahead}}$.

The selectivities of both $<, \leq$ and $>, \geq$ predicates are equivalent. For each tuple pair $\mathsf{t}_x, \mathsf{t}_y$ that satisfies the predicates $\mathsf{p}_1 : \mathsf{t}_x.\mathsf{A}_i < \mathsf{t}_y.\mathsf{A}_j$ (and its implied predicates $\mathsf{p}_{1\Rightarrow}$), there is the tuple pair $\mathsf{t}_y, \mathsf{t}_x$ that satisfies the predicates $\mathsf{p}_2 : \mathsf{t}_x.\mathsf{A}_i > \mathsf{t}_y.\mathsf{A}_j$ (and its implied predicates $\mathsf{p}_{2\Rightarrow}$). The selectivity of a predicate $\bar{\mathsf{p}}$ is given simply by subtracting the selectivity of p from the total number of tuple pairs. Out of the 30 tuple pairs in the *employees* instance, only 6 tuple pairs satisfy the predicate $\mathsf{t}_x.\mathsf{Name} = \mathsf{t}_y.\mathsf{Name}$, but $30 - 6 = 24$ tuple pairs satisfy the predicate $\mathsf{t}_x.\mathsf{Name} \neq \mathsf{t}_y.\mathsf{Name}$.

Let us assume uniform distribution of attribute values and high attribute cardinality (i.e., number of distinct values). The predicates with operators $(\neq, <, \leq, >, \geq)$ have low selectivity compared to equality predicates $(=)$. Framing $\mathsf{e}_{\mathsf{ahead}}$ to hold inequality predicates $(\neq)$ minimizes the number of inconsistent evidence, and therefore, the number of evidence reconstruction required. We can choose whether $\mathsf{e}_{\mathsf{ahead}}$ should hold $<, \leq$ or $>, \geq$ predicates without increasing the number of reconstructions. The evidence $\mathsf{e}_{\mathsf{ahead}}$, however, should not include both $<, \leq$

and $>, \geq$ predicates because that would only increase the number of inconsistent evidence. If $\mathsf{e}_{\mathsf{ahead}}$ holds $<, \leq$ predicates, then array B must be reconstructed for the correspondent $>, \geq$ predicates, or vice versa. Reconstructing single evidence requires accessing the array of evidence B and performing simple set operations. Because array B reflects the quadratic tuple pair space, minimizing the number of evidence reconstruction considerably reduces the overall runtime.

Evidence ahead initialization. For ease of exposition, let $\mathsf{e}_{\mathsf{ahead}}$ denote a general evidence that includes every predicate $\mathsf{p} \in \mathsf{P}$ such that $\mathsf{p.o} \in \{\neq, <, \leq\}$. DCFINDER initializes an evidence array B of size $|\mathsf{r}| \cdot |\mathsf{r}|$, and instantiate every element of B as a copy of $\mathsf{e}_{\mathsf{ahead}}$. We next describe how the algorithm reconstructs the array B for predicates with operators $\{=, >, \geq\}$, so that B represents a consistent state with regard to the predicate space and dataset tuple pairs. These procedures can be straightforwardly adjusted to use other settings of the evidence $\mathsf{e}_{\mathsf{ahead}}$.

3.5.2 Evidence reconstruction

DCFINDER uses PLIs to find the inconsistent tpid's of B, and then iterates over those elements to perform evidence reconstructions. We can find inconsistent tpid's from combinations of ordered pairs (l_1, l_2). The procedures to define and combine pairs of tuple identifiers (l_1, l_2) are based on the types of each predicate.

Consider the case for predicates of the form $\mathsf{p}\colon \mathsf{t}_x.\mathsf{A}_i = \mathsf{t}_y.\mathsf{A}_i$. Recall that PLIs are sets of clusters $c = \langle k, l \rangle$, and each cluster c keeps track of all tuples identifiers l with the same value k. From each cluster $c = \langle k, l \rangle \in \Pi(\mathsf{A}_i)$, DCFINDER builds ordered pairs (l_1, l_2), where $l_1 = l$ and $l_2 = l$. The tuple pairs with $t_x \in l_1$, $t_y \in l_2$, and $t_x \neq t_y$ are precisely those tuple pairs that satisfy the equality predicate p. Each of these tuple pairs is assigned a tpid (Equation 3.1), which is stored in an ordered set $\mathbf{T}$. Consider the cluster $\langle \text{DBA}, \{3,4\} \rangle$ of $\Pi(\text{Position})$ for instance. It gives us the ordered pair $(\{3,4\}, \{3,4\})$, and therefore, tuple pairs $\mathsf{t}_3, \mathsf{t}_4$ and $\mathsf{t}_4, \mathsf{t}_3$. These are exactly some of the tuple pairs that satisfy the predicate $\mathsf{p}_5 : \mathsf{t}_x.\text{Position} = \mathsf{t}_y.\text{Position}$. From Equation 3.1, and tuple pairs $\mathsf{t}_3, \mathsf{t}_4$ and $\mathsf{t}_4, \mathsf{t}_3$, we get tpid's 22 and 27. These tpid's point to evidence in the array B that are incorrectly holding the predicate $\mathsf{p}_6 : \mathsf{t}_x.\text{Position} \neq \mathsf{t}_y.\text{Position}$, so we must reconstruct these pieces of evidence to hold p_5 instead. Following the above procedures for every cluster of $\Pi(\text{Position})$ gives us every piece of evidence we must reconstruct for predicate p_5.

Finding tuple pairs that satisfy other types of predicates follows a similar principle, but with a slight change on how ordered pairs (l_1, l_2) are arranged. The procedure for predicates on different attributes, $\mathsf{p}\colon \mathsf{t}_x.\mathsf{A}_i = \mathsf{t}_y.\mathsf{A}_j$ where $i \neq j$, is as follows: For each cluster $c = \langle k, l \rangle \in \Pi(\mathsf{A}_i)$, DCFINDER probes $\Pi(\mathsf{A}_j)$ for a cluster $c' = \langle k', l' \rangle \in \Pi(\mathsf{A}_j)$ such that $k = k'$. If there is a match, DCFINDER builds an ordered pair (l_1, l_2), where $l_1 = l$ and $l_2 = l'$. Building the tuple pair representation from (l_1, l_2) follows the same principle described before. Finally, the procedure for greater-than predicates with the form $\mathsf{t}_x.\mathsf{A}_i > \mathsf{t}_y.\mathsf{A}_j$ is as follows. For each cluster $c = \langle k, l \rangle \in \Pi(\mathsf{A}_i)$, DCFINDER looks up every cluster $c' = \langle k', l' \rangle \in \Pi(\mathsf{A}_j)$ such that $k > k'$. For each match, a new ordered pair (l_1, l_2) is built. DCFINDER transforms these tuple

pair representations into the tpid's, just as described before. The algorithm keeps a map $\mathscr{T}$ of associations between a predicate p and the ordered set of tuple pair identifiers that satisfy p.

Algorithm 1 shows the steps to find all the tuple pair identifiers that point to inconsistent evidence in array B, given a predicate space and relation instance. DCFINDER calculates tuple pair identifiers only for $\{=,>\}$ predicates. By minding the implication property, the algorithm reconstructs B for $\{\geq\}$ predicates as well.

Algorithm 1: Find the identifiers of inconsistent tuple pairs

Data: Relation instance r, and predicate space P
Result: A mapping $\mathscr{T}$ from predicates to tuple pair identifiers

```
for A_i ∈ R do
    build PLI Π(A_i)
    if A_i is numeric then
        sort Π(A_i) in descending order of keys k
𝒯 ← ∅
foreach p ∈ P where p.o ∈ {=,>} do
    Use PLIs to compute T of p
    𝒯{p} ← T
return 𝒯
```

Algorithm 2 shows how DCFINDER materializes and reconstructs tuple pairs evidence. Evidence array B is initialized with copies of $\mathsf{e}_{\mathsf{ahead}}$. For each pair $\langle \mathsf{p}, \mathbf{T} \rangle$ in the mapping $\mathscr{T}$, DCFINDER performs a sequence of reconstructions. Given a tpid set $\mathbf{T}$, the algorithm updates $B[\mathsf{tpid}]$ for each $\mathsf{tpid} \in \mathbf{T}$. The operations slightly differ from each other depending on the type of the predicate.

Algorithm 2: Materialization and reconstruction of evidence

Data: Mapping $\mathscr{T}$, relation instance r, and predicate space P
Result: Evidence array B

```
e_ahead ← every p ∈ P where p.o ∈ {≠,<,≤}
initialize array B, each element is a copy of e_ahead
foreach p ∈ P where p.o ∈ {=,>} do
    fix ← build predicate mask of p
    foreach tpid ∈ 𝒯{p} do
        B[tpid] ← B[tpid] ⊕ fix
return B
```

For now, let p be a *non-numeric* equality (=) predicate, and $B[\mathsf{tpid}]$ an evidence we need to reconstruct for p. At this stage, $B[\mathsf{tpid}]$ holds the inequality complement ($\neq$) $\bar{\mathsf{p}}$ of p. But we want $B[\mathsf{tpid}]$ to hold p, not $\bar{\mathsf{p}}$. Let fix denote a predicate set that includes both p and $\bar{\mathsf{p}}$, that is, $\mathsf{fix} \leftarrow \{\mathsf{p}, \bar{\mathsf{p}}\}$. The symmetric difference[1] between $B[\mathsf{tpid}]$ and fix, denoted as

[1]The symmetric difference is implemented as a simple exclusive or operation (XOR).

$B[\text{tpid}] \leftarrow B[\text{tpid}] \oplus \text{fix}$, gives us a consistent $B[\text{tpid}]$ with regard to p. If p is a *numeric* equality ($=$) predicate, fix must also include the correspondent $<, \geq$ predicates of p. Once the symmetric difference has been applied, $B[\text{tpid}]$ satisfies p and its correspondents $\leq, \geq$. That fulfills the implication requirement for p.

Finally, let p be a greater than ($>$) predicate, and an evidence $B[\text{tpid}]$ be inconsistent for p. $B[\text{tpid}]$ holds the correspondent $\{\neq, <, \leq\}$ predicates of p, but should hold $\{\neq, >, \geq\}$ predicates, instead. To reconstruct $B[\text{tpid}]$, we need to set fix to hold $\{<, \leq, >, \geq\}$ and calculate the symmetric difference $B[\text{tpid}] \leftarrow B[\text{tpid}] \oplus \text{fix}$. This operation removes the correspondent $\{<, \leq\}$ predicates of p, but includes the correspondent $\{>, \geq\}$ ones. Figure 3.4 illustrates part of the reconstruction for the evidence of *employees* with regard to the inequalities predicates on attribute Hired. The cluster $\langle 2012, \{0,5\}\rangle$ pairs with cluster $\langle 2010, \{1,2,3,4\}\rangle$ to form tpids $1, 2, 3, 4, 31, 32, 33, 34$. These elements initially hold p_{15} and p_{16}, but are reconstructed to correctly hold p_{17} and p_{18}.

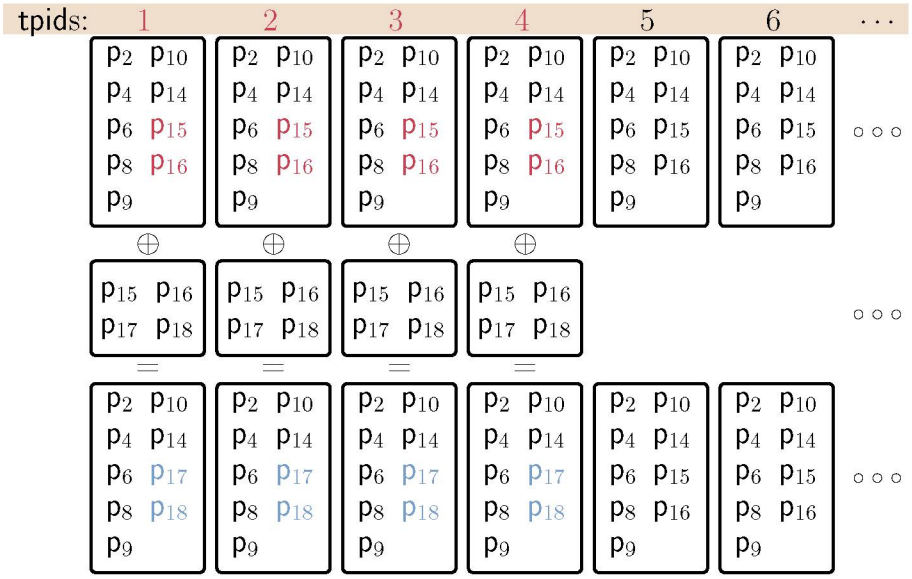


Figure 3.4: Part of the reconstruction for the evidence of *employees* and predicates $\text{p}_{17} : \text{t}_x.\text{Hired} > \text{t}_y.\text{Hired}$ and $\text{p}_{18} : \text{t}_x.\text{Hired} \geq \text{t}_y.\text{Hired}$.

3.5.3 How to scale up to large datasets

Storing arrays of evidence B incurs a quadratic space overhead in the number of tuples because each array B represents evidence of all tuple pairs. Also of quadratic space are the sets of tuple pair identifier $\mathbf{T}$ used to reconstruct B because they grow as a function of the number of tuple pairs. Storing all the data of B and $\mathbf{T}$ at once may be infeasible as it can sooner or later exhaust any memory limit. It turns out that slightly modifying how these structures are built enables DCFINDER to scale up for larger datasets. DCFINDER uses a multi-level partitioning scheme based on the range of tuple pair identifiers. The idea is to create a partial evidence set for each range, and then merge these sets into the final and correct evidence set. The scheme enables DCFINDER to: (i) handle larger relation instances, and (ii) use multiple parallel threads.

Figure 3.5 illustrates the partitioning scheme. Instead of materializing whole sets of tuple pair identifiers **T**, DCFINDER processes only fractions of **T** at a time. Virtual sets of tuple pair identifiers **T** are partitioned into chunks $\mathbf{T} = \{T_0, T_1, \ldots, T_s, \ldots\}$, $s \in \mathbb{N}$. Partitioning is based on the disjoint ranges of tpid values. Assuming a maximum chunk length ω, chunk T_0 can store any $\mathsf{tpid} \in [0, \omega)$, $\mathsf{tpid} \in \mathbb{N}$. Chunk T_s can store any $\mathsf{tpid} \in [\mathsf{low}, \mathsf{high})$, where $\mathsf{low} = s \cdot \omega$, and $\mathsf{high} = (s+1) \cdot \omega$. In a similar fashion, DCFINDER processes all the evidence of B using small evidence fragments. Each fragment stores at most λ evidence elements. This two-tier partitioning scheme benefits from data locality, as we show in our experimental evaluation.

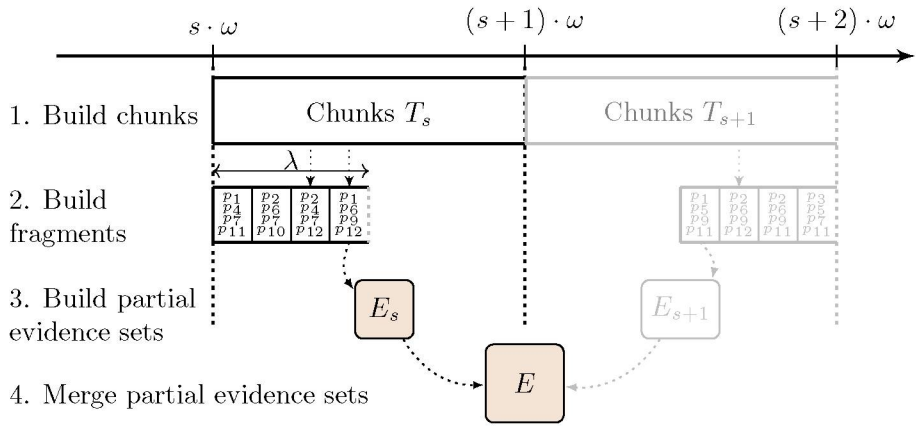


Figure 3.5: Evidence set building: Partitioning of tuple pair identifiers into chunks, and splitting of tuple pair evidence into evidence fragments.

Let us consider the s-th run. We build tuple pair identifier sets T_s for every predicate required to materialize the evidence set. We want each chunk T_s to hold every tpid associated to **T** such that $\mathsf{low} \leq \mathsf{tpid} < \mathsf{high}$. Recall that tuple pair identifiers tpids are drawn from ordered pairs $\langle l, l' \rangle$. DCFINDER shrinks pairs $\langle l, l' \rangle$ so they yield tpids within the range of chunk T_s. From Equation 3.1, we see that any tuple pair $\mathsf{t}_x, \mathsf{t}_y$ such that $\mathsf{t}_x \in l$, and $\mathsf{t}_x > \mathsf{high}/|\mathsf{r}|$ yields a tpid that is greater or equal to high, and therefore $\mathsf{t}_x, \mathsf{t}_y$ falls outside the range of T_s. Depending on the size of chunks and relation instances, other $\mathsf{t}_x, \mathsf{t}_y$ settings may also yield tpids outside the range of T_s. DCFINDER removes such tuple settings from ordered pairs $\langle l, l' \rangle$. Any tpid from $\langle l, l' \rangle$ is guaranteed to fall within the range of T_s after $\langle l, l' \rangle$ has been shrunk. DCFINDER proceeds to reconstruct evidence after all chunks T_s are created.

DCFINDER follows Algorithm 2, but reconstructs small evidence fragments instead of the potentially huge evidence array B. The algorithm initializes a fragment using $\mathsf{e}_{\mathsf{ahead}}$. Then it iteratively consumes tpids from chunks to perform the reconstructions. It stops consuming tpids if a tpid is no longer within the fragment range. The current fragment is consistent after all chunks within the same range have been processed. DCFINDER then iterates over the evidence of the current fragment to retain two information: (i) distinct evidence, and (ii) evidence multiplicity. Evidence of reflexive tuple pairs, i.e., $\{\mathsf{t}_x, \mathsf{t}_x\}$, are skipped. The evidence set produced at that point is partial, because it regards only tuple pairs within a given range. DCFINDER requires an additional step to merge all partial evidence sets. As discussed before, the number of distinct

evidence is very small compared to the number of total tuple pairs. Thus, merging partial evidence sets does not incur significant overhead.

The primary computational pattern for evidence reconstruction is the sequential read of chunks followed by symmetric difference computations. If the chunk is too small, the number of runs increases. On the other hand, if the chunk is too large, memory may end up exhausted. The symmetric difference operation is implemented as an XOR operation, which is usually optimized in modern CPU architectures. Because DCFINDER needs to perform many of these operations, improving data locality helps reducing cache miss penalty. We performed micro-benchmarks to verify the influence of chunk size ω and fragment size λ parameters in runtime. Our experiments (Section 3.7) show that using relatively small evidence fragments decreases cache misses, and thus improve runtime. We observed that settings where the fragment size is just a fraction of the chunk size yields better runtime than the settings where the size of chunks and fragments are the same.

Keeping a simple counter for each distinct evidence suffices, so we are able to accommodate the cover search (Section 3.6) to discover approximate denial constraints. The final evidence set E is a simple hash map with evidence as keys, and evidence frequency as values. We use $\mathsf{counter}$ to denote a function $\mathsf{E} \to \mathbb{N}$ such that $\mathsf{counter}(\mathsf{e})$ returns the frequency of evidence e.

DCFINDER can build partial evidence sets independently of each other, because chunks $\{T_0, T_1, \ldots, T_s, \ldots\}$ are disjoint. It picks up available threads from a thread pool to serve as workers. The only data shared across workers is the data from PLIs, and from the final evidence set. Multiple workers can safely read PLIs because they never change once built. Each worker operates on its own chunks and fragments to generate its partial evidence set. The concurrent access to the final evidence set is synchronized via latches. This last operation does not impose significant overhead: most time is spent finding the inconsistent tpids and fixing pieces of evidence. As we show in Section 3.7, the evidence set building phase of DCFINDER scales (almost) linearly in the number of CPU cores.

3.6 DENIAL CONSTRAINT SEARCH

This section describes how DCFINDER uses the evidence set to discover minimal approximate (and exact) denial constraints. It also describes three measures to score the interestingness of the discovered denial constraints.

3.6.1 Minimal covers

A denial constraint can be any subset of the predicate space P, so entirely traversing the search space with $2^{|\mathsf{P}|}$ candidates is infeasible. Discovering attribute dependencies is likely an intractable problem [132, 133]. For example, the authors of [133] have recently shown that detecting functional dependencies is a $W[2]$-complete problem. The result directly impacts the computational hardness of denial constraint discovery, because denial constraints subsume

functional dependencies. Despite their computational complexity, data profiling algorithms have managed to perform quite well on various real-world datasets [29, 91, 92].

The problem of discovering all minimal denial constraints can be transformed into the problem of finding all *minimal covers* of the evidence set [32]. The latter problem is cognate with other problems, such as enumerating *hitting sets* or *hypergraph traversals* [134]. These problems have been studied under a variety of domains for their wide range of applications [134, 135]. We make use of the approach of [32], because it easily accommodates the search of approximate (partial) covers, and therefore, approximate denial constraints. The approach works well in practice, as discussed in Section 3.7.

An evidence $\mathsf{e} \in \mathsf{E_r}$ cannot hold predicates $\{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$ and $\{\bar{\mathsf{p}}_1, \ldots, \bar{\mathsf{p}}_m\}$ simultaneously. If e holds $\{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$, any denial constraint φ containing at least one predicate of $\{\bar{\mathsf{p}}_1, \ldots, \bar{\mathsf{p}}_m\}$ could not be violated by the tuple pairs that yield evidence e. For φ to be exact, that intuition must apply for every evidence $\mathsf{e} \in \mathsf{E_r}$. That is why we find covers of the full evidence set $\mathsf{E_r}$. A cover Q_1 is a set of predicates that intersects with every evidence of $\mathsf{E_r}$, i.e., $\forall \mathsf{e} \in \mathsf{E_r}, Q_1 \cap \mathsf{e} \neq \emptyset$. The cover Q_1 is minimal if there does not exist a Q_2 that is a strict subset of Q_1 and intersects with the same elements of Q_1, i.e., $\nexists Q_2 \subset Q_1$ such that $\forall \mathsf{e} \in \mathsf{E_r}, Q_2 \cap \mathsf{e} \neq \emptyset$. The following theorem holds for discovering denial constraints (see [32] for proof).

Theorem 1. A denial constraint φ: $\neg(\bar{\mathsf{p}}_1 \wedge \ldots \wedge \bar{\mathsf{p}}_m)$ holds in relational instance r if the set Q: $\{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$ is a cover of the evidence set $\mathsf{E_r}$. The denial constraint φ is minimal if Q is minimal.

In addition, we must be able to discover approximate denial constraints. Recall that the degree of approximation ε of a denial constraint φ is based on the number of tuple pairs that do not satisfy φ. The multiplicity of an evidence set is given by $\|\mathsf{E}\| = \sum_{\mathsf{e} \in \mathsf{E}} \mathsf{counter}(\mathsf{e})$, that is, how many tuple pairs yielded all evidence of E. The multiplicity $\|\mathsf{E}\|$ is equal to $|\mathsf{r}| \cdot (|\mathsf{r}| - 1)$ if $\mathsf{E} = \mathsf{E_r}$. Consider again the set Q: $\{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$, but assume that E is only a subset of the full evidence set $\mathsf{E} \subseteq \mathsf{E_r}$ such that $\forall \mathsf{e} \in \mathsf{E}, Q \cap \mathsf{e} = \emptyset$. The set Q approximately covers the full evidence set $\mathsf{E_r}$ if $\|\mathsf{E}\| \leq \varepsilon \cdot |\mathsf{r}| \cdot (|\mathsf{r}| - 1)$. If so, the predicate set Q is an ε-approximate cover of $\mathsf{E_r}$, and it is minimal if there does not exist a strict subset of Q that is also an ε-approximate cover of $\mathsf{E_r}$.

Algorithm 3 presents the minimal cover search. It is a heuristic-based depth-first search for which nodes are recursively formed based on evidence set coverage. Each node maintains a path of the search tree $Q \subseteq \mathsf{P}$, the set of evidence not covered by the current path $\mathsf{E}_{path} \subseteq \mathsf{E}$, the set of predicates that can be included in further branches $\mathsf{P}_{path} \subseteq \mathsf{P}$, and all minimal covers MC found in prior branches. Every path is a cover candidate. At first $Q = \emptyset$, $\mathsf{E}_{path} = \mathsf{E_r}$, $\mathsf{P}_{path} = \mathsf{P}$, and $\mathsf{MC} = \emptyset$. To unfold a new branch, the algorithm adds a predicate p_{add} to the new path and updates the information for the child node. The child evidence set E_{new} is the result of removing all evidence that contain p_{add} from the parent evidence set E_{path}. The child predicate set P_{new} is every predicate $p \in \mathsf{P}_{path}$ such that $\mathsf{p} \not\sim \mathsf{p}_{add}$.

Algorithm 3: Find Minimal Covers [32]

Data: Evidence set $\mathsf{E_r}$, Predicate space P, Error threshold ε
Result: Set of minimal covers MC

```
1  MC ← ∅
2  findCover (∅, E_r, P, MC)
3  Function findCover (Q, E_path, P_path, MC)
4      if ||E_path|| ≤ ε · |r| · (|r| − 1) then
5          if no subset of size |Q| − 1 of Q ε-covers E_r then
6              MC ← MC ∪ Q
7          return
8      else if P_path = ∅ then
9          return
10     else
11         sort P_path based on tuple pair coverage of E_path
12         for p_add ∈ P_path do
13             Q ← Q ∪ p_add
14             if Q is implied by MC then
15                 Q ← Q \ p_add
16                 continue
17             E_new ← {e | e ∈ E_path and p_add ∉ E_path}
18             P_new ← {p | p ∈ P_path and p ≁ p_add}
19             findCover (Q, E_new, P_new, MC)
```

Two base cases stop the recursion. First, the algorithm finds an approximate cover if the path Q removes large pieces of evidence of E_r such that $\left\|\mathsf{E}_{path}\right\| \leq \varepsilon \cdot |\mathsf{r}| \cdot (|\mathsf{r}| - 1)$. Consequently, the corresponding denial constraint of Q could be violated by no more than $\left\|\mathsf{E}_{path}\right\|$ tuple pairs. If $\mathsf{E}_{path} = \emptyset$, Q is an exact cover. To ensure minimality, the algorithm tests whether there exists an immediate subset of Q that also (approximately) covers E_r. If it does not find such a subset, the predicate set Q is added to the result MC as a minimal cover. Second, if the search reaches a node for which there are still enough evidence to cover, but there are no predicates to form new branches, then there is no valid cover in that branch.

The tuple pair coverage of a predicate p is the multiplicity of the evidence set in which all evidence contain p, that is, $\|\mathsf{E}\|$ such that $\mathsf{e} \in \mathsf{E}$ and $\mathsf{p} \in \mathsf{e}$. The heuristic to unfold new paths is to include predicates in dynamic ordering of tuple pair coverage. The search adds predicates satisfied by most tuple pairs first, i.e., those predicates that reduce the evidence set size the most. Removing predicates from E_{new} changes the tuple pair coverage distribution for the remaining candidate predicates P_{new}, so the algorithm needs to compute a new predicate ordering for each new branch. The sooner the evidence set becomes small enough, the sooner the algorithm finds minimal covers. The algorithm uses these covers MC to reduce the number of searches. Before updating the information for a new path (E_{new} and P_{new}), the algorithm checks if that path is already in the cover. If so, there is no need to unfold that branch.

Once Algorithm 3 is finished, each minimal cover in MC is translated into a minimal denial constraint by inverting its predicates (Theorem 1). The output may contain implied denial constraints, so we need to test whether each denial constraint is implied by the remaining discovered denial constraints. This implication testing is known to be a coNP-complete problem [56]. The authors of [32] introduced an inference system for denial constraints and describe an algorithm to test denial constraint implication with it. We use this algorithm to remove implied denial constraints from the output of all denial constraint algorithms. Although not complete, the implication testing algorithm is correct and helps to remove many implied denial constraints from the output, which helps with user verification. More details on the static analyses of denial constraints and other constraints can be found in [56, 1].

3.6.2 Interestingness measures for denial constraint

DCFINDER discovers all minimal denial constraints in a dataset. But in all likelihood, not all of them are equally useful. DCFINDER optionally estimates three interestingness measures: succinctness, coverage, and degree of approximation. We use these measures to: (i) pruning denial constraint candidates that fall beneath interestingness thresholds, and (ii) ranking denial constraints to help users selecting relevant ones.

Succinctness has been used to rank denial constraints in [32]. It is inversely proportional to the number of distinct symbols (attributes and operators) in the predicates of a denial constraint: the fewer symbols a denial constraint has, the more succinct it is. The measure is based on the minimum description length principle: data representations with fewer symbols are more succinct. DCFINDER can use succinctness to prune denial constraints during cover search. To do so, it simply counts how many symbols a candidate denial constraint expresses before checking it. If the quantity is greater than a given threshold, there is no need to check further paths from that candidate denial constraint—the succinctness can only decrease.

Coverage is described in [32] as the statistical significance of a denial constraint based on the proportion of tuple pairs that satisfy a given set of predicates. It is given by a weighted sum of tuple pairs scores. Given a denial constraint φ with $|\varphi|$ predicates, each tuple pair scores the denial constraint φ based on how many predicates that tuple pair satisfies. The larger the amount of tuple pairs satisfying a number of predicates close to $|\varphi| - 1$, the higher the coverage of φ. There is no guarantee that coverage always decreases for a given path, so we used this measure only during post-processing to rank denial constraints according to their coverage scores. Estimating the coverage of a denial constraint requires iterating over the evidence set and evidence frequency counters. Because many denial constraints have predicates in common to each other, this estimation can be performed in a depth-first tree traversal to save computation for denial constraints sharing a common prefix.

We can additionally use the degree of approximation, defined in Section 3.2, to measure the interestingness of approximate denial constraints. It follows from Definition 3 that the number of tuple pairs allowed to violate an approximate denial constraint is always bounded by

the error threshold. But the number of actual violations varies between the discovered denial constraints. The degree of approximation simply shows how many tuple pairs are inconsistent with regard to an approximate denial constraint. After a minimal (approximate) cover is found, the degree of approximation is simply the multiplicity of the remainder evidence set.

3.7 EXPERIMENTAL EVALUATION

We present an experimental evaluation of DCFINDER. We used all denial constraint algorithms known to date as baselines: FASTDC [32] and BFASTDC [130] for the discovery of approximate and exact denial constraints; and HYDRA [55] for the discovery of exact denial constraints.

3.7.1 Experimental setup

We used the code provided by the authors of [55] for HYDRA and FASTDC. The code of BFASTDC was provided by the authors of [130]. We implemented DCFINDER from scratch. All implementations were written in Java and run in main memory after dataset loading. We integrated all implementations with the data profiling framework Metanome [136] to guarantee a unified testing environment. To keep consistent comparisons, we set all algorithms to replace NULL values with default values (i.e., empty strings for non-numeric attributes, or $-\infty$ for numeric attributes). This approach has been used also in the implementations of [55].

The strategies that FASTDC, BFASTDC and DCFINDER use to build evidence sets are designed to run over multiple threads. Therefore, unless stated otherwise, the reports for these algorithms are from multi-thread executions. The authors of [55] do not present a parallel version of HYDRA, so we use the implementation of the algorithm just as it is described in the paper. In addition, we implemented a new version of HYDRA, namely HYDRA+, so the algorithm can benefit from parallel execution in its systematic tuple pair sampling phase. This parallel step is implemented in similar fashion to the grid scheme used in FASTDC.

The experiments were run on an Intel Core i7-7700HQ machine (2.8 GHz, 4 physical cores/8 logical cores, 32 KB for L1, 256 KB for L2, and 6 MB for shared L3); 16 GB RAM; 256GB SSD; Ubuntu 16.04; and Java 1.8 with the JVM heap space limited to 8 GB. The runtime reports are the average measurement of three independent runs.

Table 3.2 shows the main characteristics of the datasets used in our experiments. The majority of these datasets have been used in related work. The *Hospital* and *Tax* datasets have been used to evaluate denial constraint discovery algorithms in [32, 55]. The *Adult*, *Flight*, and *NCVoter* datasets have been used to evaluate FD discovery algorithms in [91]. The *Inspection* dataset has been used to evaluate data cleaning systems in [34]. We additionally used the *Airport*

dataset, which contains a list of airport codes and locations. The following page provides the implementation of our algorithm and pointers to all datasets[2].

Table 3.2: Datasets used to evaluate the denial constraint discovery algorithms.

Name	Type	#tuples	#attributes	#predicates
Adult	real-world	32,561	15	54
Airport	real-world	55,113	18	48
Flight	real-world	500,000	20	88
Hospital	real-world	114,919	15	44
Inspection	real-world	170,000	19	74
NCVoter	real-world	938,085	22	60
Tax	synthetic	100,000	15	58

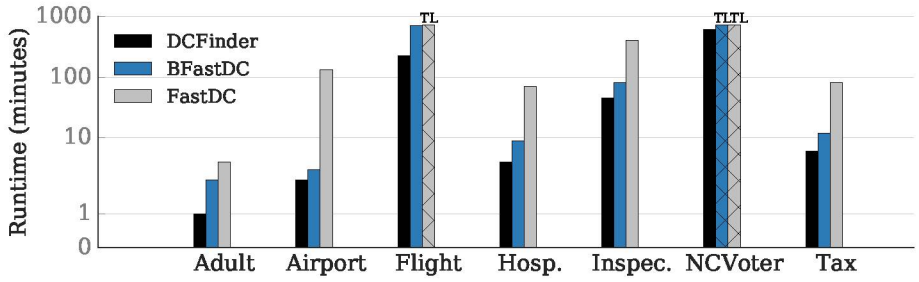
3.7.2 Discover of approximate denial constraints

We ran DCFINDER , FASTDC, and BFASTDC for all datasets shown in Table 3.2. We used degrees of approximation $\varepsilon = 0.01$ and $\varepsilon = 0.05$; these values have been previously used to evaluate the discovery of approximate dependencies [91]. We set the chunk and fragment lengths of DCFINDER to 5×10^6 and 5×10^3, respectively. We evaluate varying chunk and fragment lengths in Section 3.7.6, and varying degrees of approximation in Section 3.7.7.

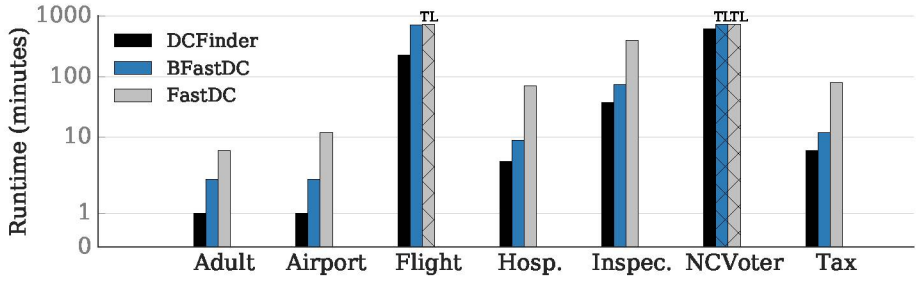
The results in Figure 3.6 show that DCFINDER is the fastest algorithm among the competitors. For *Tax* and *Hospital*, DCFINDER is at least 2× as fast as BFASTDC, and at least 13× times faster than FASTDC. The performance gains of our algorithm is higher for larger datasets. Using a degree of approximation $\varepsilon = 0.01$, for instance, DCFINDER took approximately 228 minutes to process *Flight*, BFASTDC took nearly 715 minutes, but FASTDC could not finish within the time limit of 12 hours. DCFINDER was the only algorithm able to process *NCVoter* within the time limit. The three algorithms use the same minimal cover search strategy; thus, the difference in their performance is a reflection of how efficiently they build evidence sets. Here, a good efficiency indicator is tuple pair throughput; i.e., how many tuple pairs an algorithm processes in a fixed amount of time. DCFINDER achieved better throughput than the competitors, especially for large datasets. This shows that, in terms of performance, DCFINDER improves the state of the art for the discovery of approximate denial constraints.

The algorithms discovered the largest sets of denial constraints in *Inspection* and *Adult*, respectively. Interestingly, the evidence sets for these two datasets were also the largest among all. With bigger evidence sets, the algorithms iterate over more evidence in each path of the cover search, which hinders runtime. For *Adult* and *Inspection*, a major part of the runtime was spent searching for minimal covers. The cover search for *Flight*, for example, was much faster than

[2] http://hpi.de/naumann/projects/repeatability/data-profiling/metanome-dc-algorithms.html



(a) Degree of approximation $g_1 = 0.01$.



(b) Degree of approximation $g_1 = 0.05$.

Figure 3.6: Runtime of approximate denial constraint discovery. The crossed bars indicate that an algorithm did not terminate within the time limit (TL) of 12 hours. The Y-axes are in log-scale.

the cover search for *Inspection*. The *Flight* dataset has a bigger predicate space, but draws an evidence set that is only a fraction (nearly a thirtieth) of the evidence set drawn from *Inspection*.

3.7.3 Discover of exact denial constraints

The next experiment focuses on the discovery of exact denial constraints; therefore, our comparisons additionally include the specialized algorithms, HYDRA and HYDRA+.

From Figure 3.7 we see that DCFINDER is faster than FASTDC and BFASTDC in every scenario. The algorithm even outperforms HYDRA and HYDRA+ in four out of seven datasets. For instance, DCFINDER was approximately 4.5× faster than HYDRA in *Airport*. But the sampling approach helped HYDRA to process some datasets faster than DCFINDER : For instance, HYDRA processed *NCVoter* approximately 3.5× faster than DCFINDER did.

DCFINDER materializes every tuple pair evidence to output evidence multiplicity, whereas HYDRA processes a fraction of tuple pairs to find only the distinct evidence. In a more detailed investigation, we found that HYDRA processed less than 0.1% of the total tuple pairs of each dataset. That is why HYDRA cannot produce the evidence multiplicity of the full dataset, which is required for discovering approximate covers, or calculating denial constraint coverage. HYDRA spent a significant amount of time correcting tuple pair samples to complete the evidence set – similar observations were made in the experimental evaluation of HYDRA. The correction was particularly efficient for datasets that draw a small evidence set, e.g., *Hospital*. However, it performed poorly for datasets with large evidence sets. HYDRA+ improved the sampling phase of HYDRA, but had minor influence on the overall runtime.

HYDRA iterates over each evidence to dynamically update the set of candidate denial constraints, so they no longer violate such evidence. The depth-first search of FASTDC, BFASTDC and DCFINDER starts from denial constraint candidates, and then updates the evidence set. Such a strategy is also penalized by large evidence sets; however, it uses the minimal covers to prune the search space as soon as they are discovered. For *Adult* and *Inspection*, the depth-first search was faster than the equivalent strategy of HYDRA. For the remaining datasets, all algorithms took less than two minutes to complete the search. This indicates that, in many cases, being able to build the evidence set in an efficient manner is crucial for the performance of the evaluated denial constraint discovery algorithms.

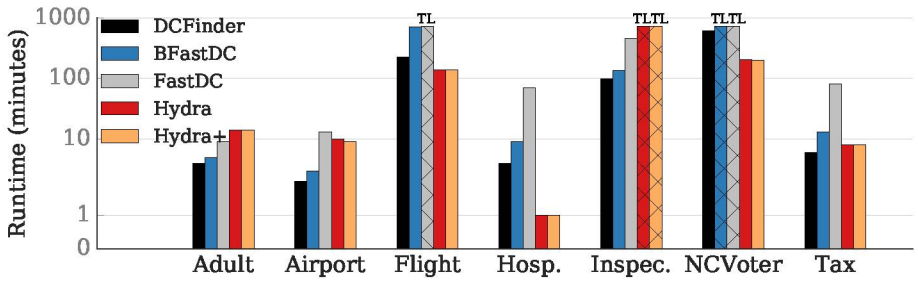


Figure 3.7: Runtime of exact denial constraint discovery. The crossed bars indicate that an algorithm did not terminate within the time limit (TL) of 12 hours. The Y-axis is in log-scale.

3.7.4 Scalability

To evaluate the scalability in the number of tuples, we started at the beginning of a dataset and incrementally added more tuples to each execution. Figure 3.8 depicts the scaling behavior for *Tax* and *Flight* datasets. All algorithms are sensitive to the number of tuples. DCFINDER , however, seems to suffer less than FASTDC and BFASTDC. The algorithm has an advantage over FASTDC because it avoids the tuple pair comparison overhead. The evidence set building strategy of DCFINDER is faster than the one of BFASTDC for two reasons. First, it does not need to calculate tpids for the inverse and implied predicates, as BFASTDC does. Second, it reduces the number of accesses to the evidence elements due to the ahead evidence allocation. For small numbers of tuples, DCFINDER may be faster than HYDRA (e.g., as in *Tax* dataset). As the number of tuples increases, Hydra starts benefiting from tuple pair sampling (e.g., when we consider more than two hundred thousand tuples for *Flight* dataset). There is an important trade-off from this improvement though: HYDRA could not be tested if we had set the degree of approximation to a value other than $\varepsilon = 0.0$. DCFINDER , on the other hand, materializes all pieces of evidence to calculate the evidence counters. That is necessary not only for discovering approximate denial constraints, but measuring the interestingness of the results based on coverage and degree of approximation.

To check scalability in the number of attributes, we began with the five initial attributes in the dataset schema. Then we incrementally added more attributes, using schema order, until every attribute of the dataset had been added. Figure 3.9 depicts the scaling behavior we obtained

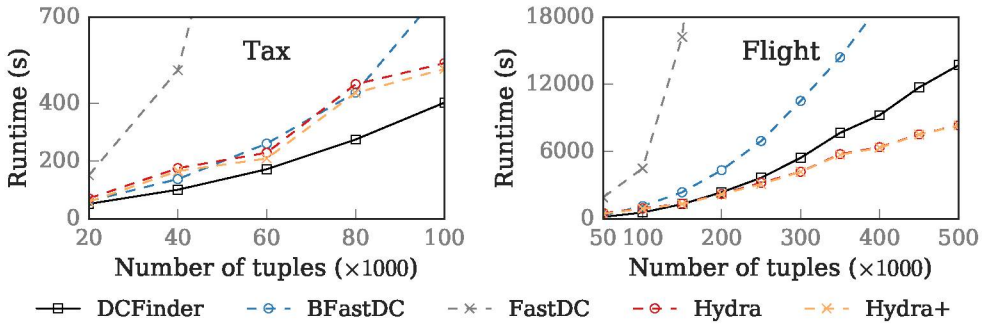


Figure 3.8: Runtime scalability in the number of rows.

for *Tax* and *Flight* datasets. We used only the first 20,000 tuples of each dataset to avoid expensive computations in the number of tuples. The runtime of all algorithms increases exponentially in the number of attributes: as the predicate space grows, so does the number of denial constraint candidates and the evidence set. Since DCFINDER , FASTDC and BFASTDC share the same cover search, the difference in their scalability is from how efficiently they build evidence sets for bigger predicate spaces. Out of these three algorithms, DCFINDER shows a slightly smoother scalability. On the other hand, FASTDC seems to have the worst performance degradation. The results in Figure 3.9 show that the performance of HYDRA is abruptly penalized when more attributes are added to its executions.

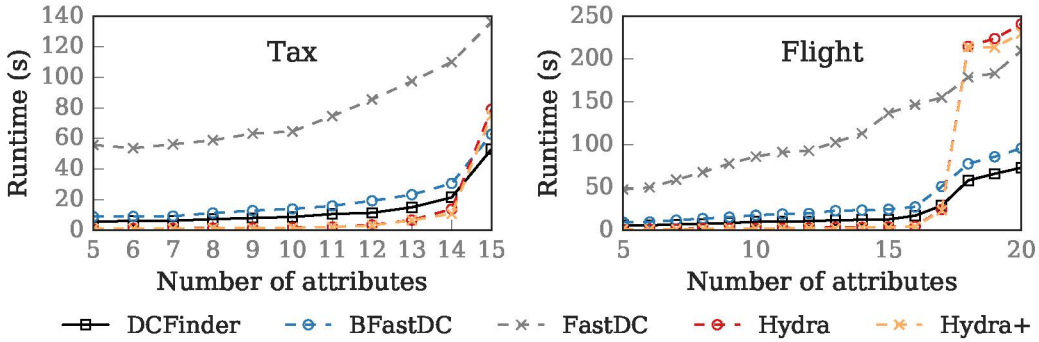


Figure 3.9: Runtime scalability in the number of attributes.

We evaluate predicate scalability using the first 20,000 tuples of *Adult*. The experiment chose different combinations of attributes at random. The goal is to check, for different combinations of predicates, how long denial constraint discovery takes and how many denial constraints are discovered. We executed the experiment twenty times and report the average values in Figure 3.10. As expected, the predicate scaling of all algorithms behaves in a similar way to their attribute scaling. Just as there is exponential growth in runtime, there is exponential growth in the number of denial constraints.

3.7.5 Memory consumption

The next experiment measures how much memory is required by the different denial constraint discovery algorithms. For the largest datasets, *Flight* and *NCVoter*, we executed

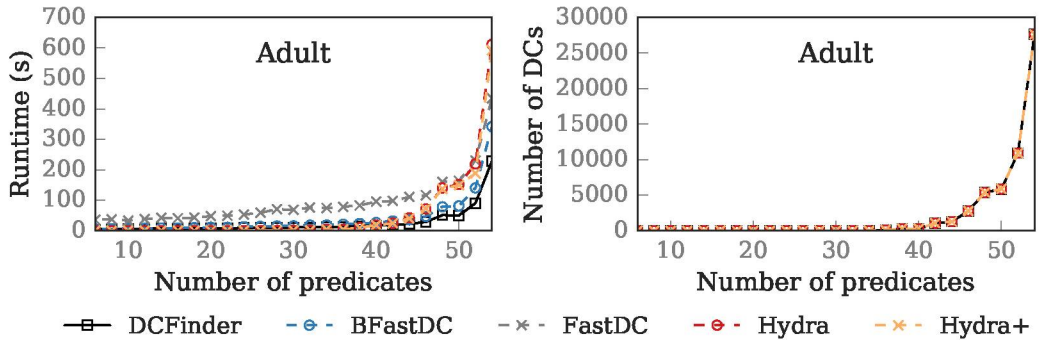


Figure 3.10: Runtime scalability in the number of predicates.

each algorithm using a maximum heap size of 64MB. Then, we repeatedly doubled this value until the respective algorithm was able to actually process that dataset (up to the time limit for slower algorithms). All algorithms had similar memory footprints. To process *Flight*, BFASTDC required 2048MB, whereas the other algorithms required 1024MB. All algorithms required 4GB to process *NCVoter*.

The main reason for this high demand is that our implementations load the full dataset into main memory to provide a fair comparison of the in-memory processing of the algorithms. This full loading incurs the overhead of encoding many attribute values as string objects. The main data structures used by DCFINDER are PLIs, chunks of tuple pair identifiers, and evidence fragments. PLIs are integer-based compact representations of datasets, and their sizes grow as a function of the number of distinct attribute values. Chunks and fragments have constant size defined by the parameters ω and λ, respectively. While these structures can be set to be as high as the available memory, we performed micro-benchmarking and found DCFINDER to perform better with relatively small values of ω and λ (as discussed in the next section).

3.7.6 DCFinder in-depth experiments

Figure 3.11 illustrates the runtime breakdown on each phase of DCFINDER . A large part of the runtime is shared between finding tpids and correcting evidence, which is expected as these phases are the core of producing accountable evidence sets. Initializing and accumulating evidence also takes a considerable amount of the runtime: This is a reflection of the quadratic complexity that the problem has in the number of tuples. For *Adult* and *Inspection*, DCFINDER spent a major part of the runtime in cover search, as explained in Section 3.7.2. The overhead from the remaining phases is relatively small compared to the overall runtime.

The next experiment focuses on the evidence set building phase of DCFINDER (Section 3.5) to highlight the scalability of DCFINDER in the number of threads. Such scaling is possible because the algorithm splits the tuple pair space into chunks, which can be processed independently of each other. The measurements are over the first 100,000 tuples of each dataset, or over the total number of tuples for *Adult* and *Airport*. Figure 3.12 shows the scalability of DCFINDER in the number of threads. The algorithm scales (almost) linearly up to the number

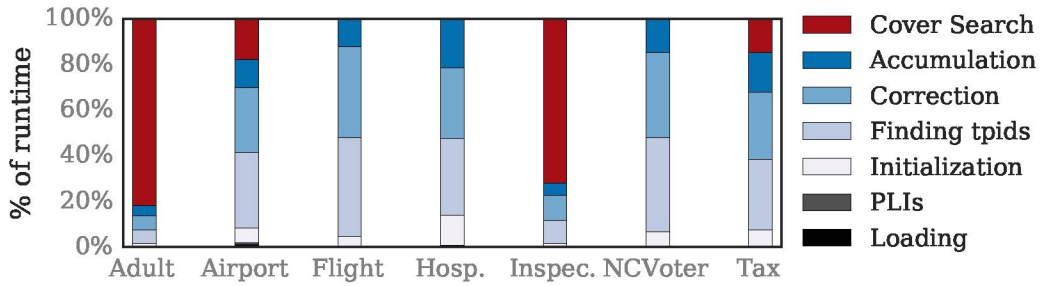


Figure 3.11: Runtime breakdown of DCFINDER ($\varepsilon = 0.01$): relative time the algorithm spent on loading datasets, building PLIs, initializing evidence, calculating tpids, correcting evidence, accumulating (hashing) evidence, and searching for minimal covers.

of physical cores (4); from there, it scales narrowly up to the number of logical cores (8). That behavior is expected as the cache resources are shared among the hyper-threads. Increasing the number of threads for more than the available logical cores does not improve runtime. Doing so is likely to increase the complexity of coordinating competing accesses to data, which may even hinder performance.

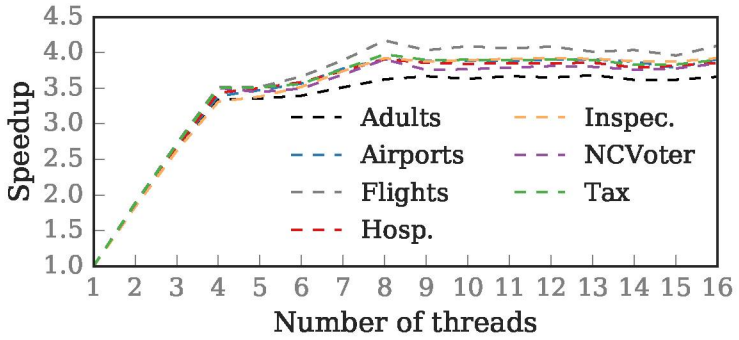


Figure 3.12: Relative runtime speedup in the number of threads (evidence set building only).

How DCFINDER splits the tuple pair space influences its efficiency. Figure 3.13 compares the behavior of the algorithm for varying sizes of chunks and fragments. We use *Tax* dataset to show this behavior, but the same trend was observed across all the evaluated datasets. The metrics of interest are runtime and cache misses (both L1 and LLC): the arrows in Figure 3.13 indicate the lowest measurements. The smaller the chunks, the more often DCFINDER iterates over PLIs to generate tpids, and the lower the tuple pair throughput (i.e., how many tuple pairs the algorithm processes in a fixed amount of time). The left plot in Figure 3.13 shows that DCFINDER runs faster as we increase chunk lengths, up until it nearly stabilizes its performance. From there, the fragment lengths at the edge (i.e., 10^2 and 10^5) negatively influenced runtime. This shows that DCFINDER is robust to the two parameters, for sizable ranges. For all datasets, DCFINDER was stable with chunk lengths around $10^6 \leq \omega \leq 10^7$ and fragments lengths at the few thousands region. After runtime inflection, the algorithm obtained no performance improvement, but increased its memory requirement.

We observed that the cache miss ratio of the settings for which DCFINDER had the best runtime was at the same level of the best cache miss ratio we measured. Recall that

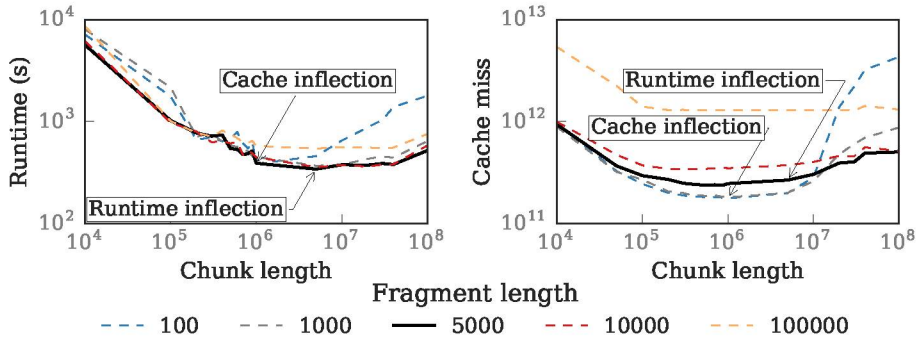


Figure 3.13: Influence of chunk and fragment length on DCFINDER runtime and cache misses. The axes are in log scale.

DCFINDER operates on two pieces of data, tpids and evidence fragments, and that it implements the correction operation as an XOR, which is directly supported by the CPU. The runtime inflection reflects a sweet-spot where DCFINDER benefits from cache locality and achieves high tuple pair throughput without exhausting main memory. We observed very small variations in the runtime inflections of the evaluated datasets. In our experiments, setting chunk length to 5×10^6 and fragment length 5×10^3 worked very well across the evaluated datasets. BFASTDC also required us to set these two parameters, so we also tried different values to tune its execution. We observed that BFASTDC works best with chunks that are slightly smaller than the chunks of DCFINDER , because BFASTDC stores tpids of all predicates of the predicate space in memory.

3.7.7 Denial constraint interestingness

The following experiment shows how different degrees of approximation impact denial constraint discovery. The approximation parameter has no influence on the evidence set building phase (for all algorithms), so we analyze only the minimal cover search behavior. We gradually increased the parameter for different executions of DCFINDER to measure how many denial constraints the algorithm returns, and how much time is spent in the minimal cover search. Figure 3.14 shows the results of these executions. The number of discovered denial constraints varies greatly between datasets. The predominant behavior is that for larger degrees of approximation the minimal cover search runs faster. The search may find approximate denial constraints sooner for larger degrees of approximation, even when there are still many evidence to cover. The number of discovered denial constraints decreases, in most cases, with larger degrees of approximation. But the number of denial constraints may also increase because discovering specializations of more general denial constraints may change the general paths followed by the cover search.

Figure 3.15 shows how DCFINDER behaves with different succinctness thresholds. We restricted the discovery to denial constraints with up to a varying number of symbols (attributes and operators). As expected, there are fewer short denial constraints—with predicates involving a few attributes and operators. This result is reflected in the cover search runtime since there are far fewer short denial constraint candidates to check. Most of the denial constraints discovered for

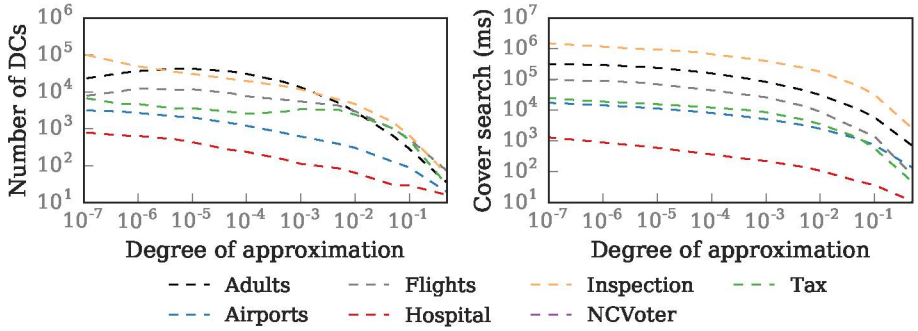


Figure 3.14: Influence of different degrees of approximation in the number of discovered denial constraints (left) and cover search time (right). The axes are in log scale.

Hospital are functional dependencies with a few attributes, therefore, increasing the succinctness threshold for this dataset did not affect the result.

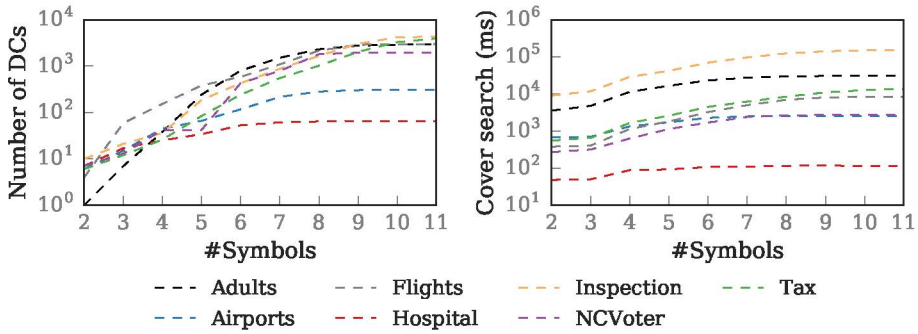


Figure 3.15: Influence of different succinctness thresholds in the number of discovered denial constraints (left) and cover search time (right). The Y axis is in log scale.

The evaluated datasets have no gold standard with a complete set of "interesting" denial constraints, so reporting the recall of the discovered denial constraints would be subjective. In an approach similar to [137], we report the precision of the top-k denial constraints. For this experiment, we used the first 50,000 tuples of each dataset. We rank all denial constraints by either coverage or succinctness, in ascending order; or degree of approximation, in descending order. Then, we empirically verify each of the top-k denial constraints to mark it as meaningful or not. The precision of each interestingness measure at k is given by the number of relevant denial constraints found in the top-k divided by k. We inspected approximate denial constraints of *Flight* and *Inspection*; and exact denial constraints of *Tax*, because of its synthetic nature. As seen in Table 3.3, the interestingness measures generally achieved good precision rates. The exception was the succinctness measure for *Inspection*, because some rules were under-fitted due to the approximate cover search.

Table 3.4 reports a sample of the discovered denial constraints. Both coverage and succinctness put the entry φ_4 at the top. The denial constraint φ_4 has no violations, and it expresses an order relationship between attributes originairportid and originairportseqid. Such a relationship is a good opportunity for query optimization. The entry φ_5 is an approximate denial constraint with relatively low succinctness, and low coverage. But because it has a small number

Table 3.3: Precision of the interestingness measures at $k = 10$.

Dataset	ε	Coverage	Succinctness	Degree of approximation
Flight	0.0001	1.0	1.0	1.0
Inspection	0.0001	0.7	0.5	0.8
Tax	0.0	0.8	0.8	-

of violations (i.e., low degree of approximation), it was straightforward to verify its correctness. The rule has a potential use for data cleaning, because it reveals problems with regard to the operating names of a company and their facility type. The denial constraint φ_6 is a meaningful business rule that did not show up at the top ranked denial constraints of *Tax*, which shows that the interestingness measures are sometimes imperfect. The denial constraint has predicates with many different symbols and, therefore, low succinctness. The more predicates a denial constraint has, the less likely a tuple pair is to add high coverage scores to that denial constraint.

Table 3.4: A sample of the discovered denial constraints.

Dataset	Denial constraint
Flight	φ_4: $\neg(t_x.\text{originairportid} \geq t_y.\text{originairportid} \wedge$ $t_x.\text{originairportseqid} < t_y.\text{originairportseqid})$
Inspection	φ_5: $\neg(t_x.\text{dbaname} = t_y.\text{akaname} \wedge t_x.\text{address} = t_y.\text{address}$ $t_x.\text{facilitytype} \neq t_y.\text{facilitytype})$
Tax	φ_6: $\neg(t_x.\text{state} = t_y.\text{state} \wedge t_x.\text{singleexemp} < t_y.\text{childexemp}$ $t_x.\text{childexemp} > t_y.\text{childexemp})$

Overall, it is possible to find relevant denial constraints by using measures of interestingness quickly. Coverage and degree of approximation are particularly useful to spot records that do not follow constraints satisfied by most of the data. The degrees of approximation and succinctness has a high impact on the runtime of cover search and in the number of discovered denial constraints. Of course, this brief analysis only scratches the surface of the problem of ranking discovered denial constraints for further use. It does show the potential, though, and the ability of DCFINDER to incorporate relevance measures to speed up execution.

3.8 SUMMARY

Motivated by the need for maintaining the consistency of data, we investigated the problem of discovering consistency rules expressed as denial constraints. We presented the DCFINDER algorithm for discovering all minimal, approximate, or exact, denial constraints of relational datasets. In DCFINDER, building a complete, but compact, evidence set is broken down into (i) creating PLIs; (ii) partitioning tuple pairs based on their ranges; (iii) preparing evidence based on predicate selectivity; and (iv) completing evidence based on PLI relationships.

DCFINDER uses evidence distribution to efficiently explore the large denial constraint search space, and to calculate two measures: the number of violations of approximate denial constraints, and the statistical significance of denial constraints based on data coverage. Our performance evaluation shows that DCFINDER is faster than all prior state-of-the-art for the discovery of approximate denial constraints. The algorithm is, at times, even faster than the algorithms specialized in discovering exact denial constraints only. Our brief study on denial constraint interestingness indicates that it is possible to quickly spot interesting denial constraints out of the many denial constraints discovered.

Chapter 4

Automatic Discovery of Reliable Denial Constraints

Data errors may appear as data outliers, duplicate records, violations of patterns (e.g., regular expressions), and violations of dependencies, i.e., data inconsistencies [138]. This chapter focuses on tackling the latter class of errors by presenting a method that helps users to choose which denial constraints they are to apply in their datasets to identify denial constraints violations.

As we discussed in Chapter 3, the automatic discovery of denial constraints from datasets is the natural alternative to designing denial constraints manually. However, there are still some barriers that limit the use of discovery algorithms in real scenarios. First, the denial constraints are as reliable as the data we use to discover them. Because obtaining 100% correct data might be infeasible, denial constraint discovery must additionally accommodate potential data errors. Second, the number of discovered denial constraints grows exponentially with the number of columns in the relation. Even if we use correct data to discover denial constraints, a great deal of the results may hold only by chance.

We introduce a method for guiding the discovery of denial constraints so that it returns results that potentially helps in data cleaning. In summary, the contributions in this chapter are the following:

- We show that the set of denial constraints discovered from clean (consistent) data typically differs from the set of denial constraints discovered from erroneous (inconsistent) data.
- We present a method to discover denial constraints that uses potentially inconsistent data to approximate the denial constraints that would be discovered in case the equivalent correct data were available. Our method selects denial constraints based on their statistical significance with regards to data distribution. We call the dependencies in such a set as *reliable* denial constraints.

- We present an experimental evaluation that shows that the set of reliable denial constraints can detect data inconsistencies with high precision and recall.

4.1 PROBLEM DEFINITION

We consider two possible versions for a relation instance r. The instance r_{clean} is complete and correct, whereas the instance r_{dirty} is any version of r_{clean} that is incomplete or incorrect. Naturally, instance r_{dirty} may contain errors and inconsistencies that are not present in instance r_{clean}. Let $\Sigma_{\mathsf{r}_{clean}}$ be the set of minimal denial constraints that hold in instance r_{clean}. By checking the records of r_{dirty} with the constraints in $\Sigma_{\mathsf{r}_{clean}}$, we find potential inconsistencies of r_{dirty}, which are detectable using the denial constraint formalism. In practice, this approach is infeasible for two reasons. First, obtaining r_{clean} is expensive, or even unrealistic. If the instance r_{clean} is not available, neither is the set $\Sigma_{\mathsf{r}_{clean}}$. Second, even if it is possible to obtain the gold instance r_{clean}, many denial constraints of $\Sigma_{\mathsf{r}_{clean}}$ may hold only by chance, therefore, not expressing any meaningful constraint. We still need to filter $\Sigma_{\mathsf{r}_{clean}}$ for meaningful denial constraints.

Our hypothesis is that it is possible to discover a set of denial constraints $\Sigma_{\mathsf{r}_{reliable}}$ that is close to the meaningful denial constraints of $\Sigma_{\mathsf{r}_{clean}}$. Nonetheless, our goal is to only use the instance r_{dirty} to do so. In particular, the denial constraints of $\Sigma_{\mathsf{r}_{reliable}}$ are expected to find real inconsistencies of instance r_{dirty}.

4.2 APPROXIMATE (BUT RELIABLE) DENIAL CONSTRAINTS

Denial constraint discovery algorithms use evidence from tuple pairs to find valid (approximate and exact) dependencies. Recall that each piece of evidence $\mathsf{e}_{\mathsf{t}_x,\mathsf{t}_y}$ is the predicate set satisfied by the pair of tuples $\mathsf{t}_x, \mathsf{t}_y$, i.e., $\mathsf{e}_{\mathsf{t}_x,\mathsf{t}_y} = \{\mathsf{p} \mid \mathsf{p} \in \mathsf{P}, \mathsf{t}_x, \mathsf{t}_y \models \mathsf{p}\}$. Different pairs of tuples may satisfy the same predicate set. In practice, the number of distinct pieces of evidence is only a fraction of the total pair of tuples of a dataset.

The evidence set E_r is the set of all evidence in r. We use $\mathsf{counter}(\mathsf{e})$ to denote the multiplicity of each piece of evidence e in E. The multiplicity of an evidence set is given by $\|\mathsf{E}\| = \sum_{\mathsf{e}\in\mathsf{E}} \mathsf{counter}(\mathsf{e})$. Each piece of evidence represents a relationship between predicates of P and the set of pair of tuples that have the same signature with regards to P. If an evidence e satisfy the predicates $\{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$, any denial constraint having at least one predicate of $\{\bar{\mathsf{p}}_1, \ldots, \bar{\mathsf{p}}_m\}$ cannot be violated by the pair of tuples that have produced the evidence e. Denial constraint discovery algorithms calculate the evidence set E_r of a dataset, then search for minimal covers of E_r. The negation of a minimal cover is a minimal denial constraint constraint.

An approximate denial constraint is the negation of a partial, minimal cover, i.e., a cover for which there still exist violating evidence. The available denial constraint discovery algorithms require a user to define the parameter ε that limits the number of violating evidence

allowed in the minimal search cover. We propose a method to set such a parameter automatically based on evidence distribution.

4.2.1 Evidence distortion

The pieces of evidence from pairs of tuples with errors are different from the equivalent pieces the equivalent pieces of evidence from the equivalent pairs of tuples having their errors fixed. Data errors degenerate the correct evidence set and the multiplicity of its elements. To illustrate this behavior, we calculate two evidence sets for a dataset called *Hospital*: $\mathsf{E}_{\mathsf{Hospital}_{clean}}$ and $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$. The details on the two versions of *Hospital* dataset are given in Section 4.3.

Figure 4.1 shows a relationship between the evidence in $\mathsf{E}_{\mathsf{Hospital}_{clean}}$ and $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$. For each evidence $\mathsf{e} \in \mathsf{E}_{\mathsf{Hospital}_{clean}}$ we plotted the multiplicity of the evidence e with regards $\mathsf{E}_{\mathsf{Hospital}_{clean}}$, and the multiplicity of the evidence e with regards $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$ (if $\mathsf{e} \in \mathsf{E}_{\mathsf{Hospital}_{dirty}}$). First, most pieces of evidence in $\mathsf{E}_{\mathsf{Hospital}_{clean}}$ intersect with the pieces of evidence in $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$. Second, there are only slight variations on evidence multiplicity. Smaller differences can be seen for the evidence with larger multiplicity, whereas more pronounced differences appear towards the tail of the plot. A few pieces of evidence from $\mathsf{E}_{\mathsf{Hospital}_{clean}}$ are not present in $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$, and a few pieces of evidence from $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$ have a considerably higher multiplicity (mainly at the tail of the plot). Besides, the set $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$ also have hundreds of spurious evidence which are not present in $\mathsf{E}_{\mathsf{Hospital}_{clean}}$. For example, one-third of the evidence of $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$ have a multiplicity of one. Nevertheless, the central tendencies of both evidence sets are significantly similar.

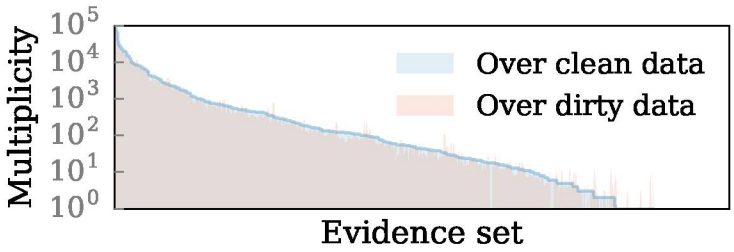


Figure 4.1: Evidence multiplicity of $\mathsf{E}_{\mathsf{Hospital}_{clean}}$, and respective $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$. X-axis is a function of the piecies of evidence of $\mathsf{E}_{\mathsf{Hospital}_{clean}}$.

The degradation in tuple pair evidence directly impacts the quality and quantity of discovered denial constraints. From the definition of approximate denial constraints (Definition 3), we observe the following.

Consider a minimal denial constraint $\varphi_1 : \forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}, \neg(\mathsf{p}_1 \wedge \mathsf{p}_2)$ of $\Sigma_{\mathsf{r}_{limpa}}$. Without loss of generality, we have two scenarios by checking a dirty instance r_{dirty} with φ_1. The first one is if r_{dirty} has no violations with regards to φ_1; therefore, denial constraint φ_1 is an exact denial constraint in r_{dirty}. The second scenario is if r_{dirty} violates φ_1; therefore, the denial constraint φ_1 is an approximate denial constraint in r_{dirty}.

In the latter scenario, discovering exact denial constraints of r_{dirty} would return a specialization of φ_1, say for example $\varphi_1' : \forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}, \neg(\mathsf{p}_1 \wedge \mathsf{p}_2 \wedge \ldots)$. When the search for

minimal covers of evidence $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$ hits the candidate with the predicates of φ_1, there is still evidence to cover. Hence, the cover search algorithm adds predicates to this candidate so it can cover the remaining evidence. Doing so masks the pairs of tuples that violate denial constraint φ_1 because its specialization φ_1' cannot find the violations anymore. Because the search is likely to cover more evidence, it is likely to reach longer paths, which increases the number of candidate denial constraints.

The number of integrity constraints a database must hold is relatively small, but the number of denial constraints discovered in production datasets can easily reach the thousands. This number comes from the denial constraint search space that exponentially grows as a function of the number of predicates in P. As we saw in Chapter 3, we can measure the semantic value of denial constraints based on a measure called coverage. It expresses the statistical significance of a denial constraint based on the weighted sum of tuple pairs scores. For a given a denial constraint φ with $|\varphi|$ predicates, each tuple pair scores the denial constraint φ based on how many predicates that tuple pair satisfies. The larger the number of tuple pairs satisfying some predicates close to $|\varphi| - 1$, the higher the coverage of φ.

The degeneration on evidence impacts both the number of discovered denial constraints and the distribution of coverage values. Figure 4.2 shows the coverages scores of the denial constraints in $\mathsf{E}_{\mathsf{Hospital}_{clean}}$ and $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$, in descending order. The number of denial constraints in $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$ is order of magnitude larger than the number of denial constraints in $\mathsf{E}_{\mathsf{Hospital}_{clean}}$. A single denial constraint of $\mathsf{E}_{\mathsf{Hospital}_{clean}}$ may have multiple specializations in $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$. The scores for these specializations reach different coverage values because the coverage estimation is based on spurious and incorrect evidence. The set $\mathsf{E}_{\mathsf{Hospital}_{dirty}}$ also produces many new denial constraints; most of them with many predicates, coverage close to zero, and without a clear meaningful semantic.

The distribution of coverage scores can be numerically seen as a set of stationary parts. The shaded areas of Figure 4.2 illustrate points of abrupt change in coverage. The number of changes is smaller and smoother in the set of denial constraints $\Sigma_{\mathsf{E}_{\mathsf{Hospital}_{clean}}}$. In addition, the set $\Sigma_{\mathsf{E}_{\mathsf{Hospital}_{clean}}}$ produces a larger number of abrupt changes, consequently, a larger number os stationary parts. The coverage classification of $\Sigma_{\mathsf{E}_{\mathsf{Hospital}_{clean}}}$ is numerically better because it shows coverages scores that are evenly distributed, with a clear separation range.

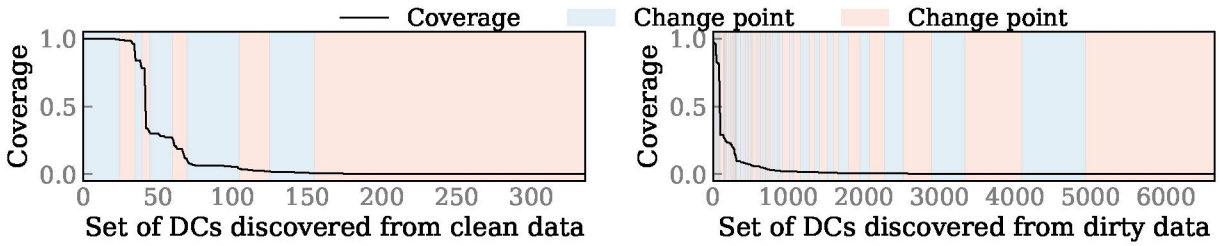


Figure 4.2: Coverage of the denial constraints in $\Sigma_{\mathsf{Hospital}_{clean}}$ (left) and $\Sigma_{\mathsf{Hospital}_{dirty}}$ (right).

4.2.2 Setting the discovery of approximate denial constraints

Our goal is to discover a set of denial constraints Σ_{conf} that is close to a subset of Σ_{clean} whose denial constraints have high coverage.

We first estimate the evidence set $\mathsf{E}_{\mathsf{r}_{dirty}}$, as the only data available is r_{dirty}. While using dirty data to produce knowledge is a challenge, it is usually safe to assume that a large percentage of data is, in fact, correct. In that case, even though data errors cause some correct evidence to fade away slowly, the central tendency is preserved. Because the variance of evidence multiplicity is high, we use the median value of evidence multiplicity as a measure for a central tendency.

Let *md* be the median value of the multiplicity of E_r. We estimate an evidence set E_{md} such that $\mathsf{E}_{md} = \{\mathsf{e} \mid \mathsf{e} \in \mathsf{E}_\mathsf{r} \wedge \mathsf{counter}(\mathsf{e}) > md\}$. If we consider only the evidence in E_{md} to discover denial constraints, we discard evidence that may be consistent with regards to predicates that are not involved in errors. For example, some tuples may contain errors in column A_i, but not in column A_j. We instead use the following formula to estimate an error threshold: $\varepsilon = 1 - \frac{\|\mathsf{E}_{md}\|}{|\mathsf{r}|\cdot(|\mathsf{r}|-1)}$. We use this estimation with a traditional approximate denial constraint discovery algorithm to guarantee that each discovered denial constraint is violated by at most $\varepsilon \cdot |\mathsf{r}| \cdot (|\mathsf{r}| - 1)$ tuple pairs. The denial constraint search is performed based on an error expectancy derived from the data itself.

The next steps are sorting the result set of denial constraints by coverage, calculating the abrupt changes in coverage scores of these denial constraints, and then returning as Σ_{conf} every denial constraint that appears before the first abrupt change.

We use a technique called *change point detection* to identify the abrupt changes [139]. Because all the coverage scores are known before-hand, we can use offline change point detection. We can think of the sorted coverage scores as a finite signal $u = \{u_1, \ldots, u_{|\Sigma|}\}$. The change point detection is to detect instants $z_1 < z_2 < \ldots < z_K$ where there are abrupt changes in u. We assume the number K of changes to be unknown. Our implementation uses a dynamic programming algorithm called pruned exact linear time (PELT) method [140]. The method does not incur any major runtime penalties, as the number of denial constraints is usually in the thousands. The method has been shown to achieve a high proportion of true changepoints, and fewer false changepoints [140, 139].

4.3 PRELIMINARY EVALUATION

We used DCfinder algorithm with the method described in Section 4.2 to discover a set of denial constraints Σ_{conf}. We measured the precision and recall of this set in finding real inconsistencies of datasets. Our prototype is a Java client connected to a PostgreSQL database. We used two datasets that have been extensively used to evaluate data cleaning systems: *Hospital* and *Flights*. The authors of [34] gently provided both clean and dirty versions of these datasets. All inconsistencies in the dirty versions are known. *Hospital* dataset has 1000 records, 20 attributes, and error rate of 0.03; *Flights* has 2376 records, 6 attributes, and error rate of 0.30.

As baselines, we use DCfinder algorithm set with error thresholds used in related work $\varepsilon = 0.01$, e $\varepsilon = 0.05$.

Table 4.1: Comparison in terms of detection of inconsistent tuple pairs.

Method	Hospital		Flights	
	Prec.	Rec.	Prec.	Rec.
Method of Section 4.2	**0.93**	**1.0**	**0.70**	**1.0**
DCFINDER with $\varepsilon = 0.01$	0.08	1.0	0.06	0.52
DCFINDER with $\varepsilon = 0.05$	0.03	1.0	0.06	0.99

Table 4.1 shows the precision and recall each method achieved. Our method is consistently better than the competitors and achieves good levels of precision and recall. Even if the error rate of a dataset is high (i.e., *Flight*), it is still able to find all the inconsistencies in the dataset. With the baseline approaches, many consistent tuple pairs are marked as incorrect, which causes the precision to decrease. Furthermore, the baseline recall is sensitive to the parameter ε, which shows that the measure must be chosen carefully. Our method uses data distribution to choose correct parameters without human intervention. Compared to the baselines, our method does not significantly increase execution times neither memory consumption. That is expected because our method only adds simple calculations on evidence multiplicity, and the change detection algorithm has linear costs.

4.4 DISCUSSION

The promising results of Section 4.3 show that it is possible to discover reliable denial constraints from inconsistent data. However, experiments need to be performed in larger scenarios: more records, attributes, and varying rates of error. Obtaining 100% correct data is a challenge, so future works shall include synthetic data to test the boundaries of our method.

Chapter 5

Efficient Detection of Data Dependency Violations

A fundamental aspect of data quality is *data consistency*. Recall the definition given by Fan: "Data consistency refers to the validity and integrity of data representing real-world entities" [1]. A natural way to capture data inconsistencies is to detect violations of *data dependencies* [1, 22]. A dependency violation is a combination of values from one or more records in the database that do not satisfy the value relationship imposed by that dependency. A database is consistent if it holds no violation of the dependencies defined for it.

As we already discussed, there has been much research on reasoning, discovery, and use of data dependencies [1, 24, 54, 22]. An important question is whether a dependency formalism is able to capture the inconsistencies commonly found in production data, i.e., its expressiveness. Early work has proposed to capture inconsistencies of traditional dependencies, such as functional dependencies and inclusion dependencies [113]; and extensions of such dependencies have been presented to overcome their expressiveness limitations [24]. Recent work has proposed to detect (and possibly repair) violations of different types of dependencies at once [54, 34]. As we saw in the previous chapters, denial constraints naturally align with such a holistic view. The formalism is one of the most general forms of dependency discussed in the database literature since it generalizes several different types of dependencies [32, 54, 34, 1]. A denial constraint expresses a set of relational predicates that specify constraints on the combination of column values. Any tuple, or set of tuples, that disagrees with these constraints is a denial constraint violation that reflects inconsistencies in the database.

The detection of denial constraint violations is an expensive operation [54, 34]. Data cleaning systems based on the formalism either rely on database management systems [34] or implement a module [54] for this task. As many legitimate denial constraints express constraints on pairs of tuples, detecting their violations exhibits a quadratic time complexity in the number of tuples [54]. This complexity is perhaps the reason the experimental evaluations of systems based on denial constraints are limited to simple dependencies (mainly functional dependencies)

or small datasets. In many real-world scenarios, however, data cleaning has to deal with large datasets and complex denial constraints.

We present VIOFINDER as our denial constraint violation detector. In summary, the contributions in this chapter are the following:

- We describe the specialized data structures that VIOFINDER uses to reduce memory overheads and enable its algorithms to perform fast operations.
- We present a customizable operator that lets us use effective algorithms to deal with complex denial constraints.
- We present an execution model that avoids materialization of large intermediates and enable optimizations inter operators.
- We provide an experimental evaluation showing that the design choices in VIOFINDER enable the algorithm to perform efficiently for several kinds of denial constraints.

The remainder of this chapter is as follows. In Section 5.1, we discuss the background and previous solutions for data dependency detection. In Section 5.2, we introduce the design of VIOFINDER and in Section 5.3 its several algorithms. Then, in Section 5.4, we present our experimental results: We compare VIOFINDER with a tool based on denial constraints and several database management systems and demonstrate that VIOFINDER is orders of magnitude faster than the competitors in many cases. In Section 5.5, we present our conclusion.

5.1 BACKGROUND AND PREVIOUS SOLUTIONS

In this section, we first present the fundamentals to represent data dependencies and data inconsistencies. Then, we review baseline approaches for the detection of data inconsistencies.

5.1.1 Denial constraints in violation detection

Denial constraints use relationships between predicates to specify inconsistent states of column values. In this chapter, we focus on denial constraints using predicates without constants, as they are computationally expensive and thus a more interesting type. We also focus on predicates over two distinct tuples, because they can express those data dependencies that are more common in practice. Nonetheless, we present an architecture and operator that can be extended to support denial constraints with other predicate forms.

Recall that for a relation to be consistent with a denial constraint φ, there cannot exist any pair of tuples such that the conjunction of the predicates of φ is true. Consider the relation *hours* in Table 5.1 and the following constraint: For any two employees with the same role, the one who has worked more hours should not receive a lower bonus than the other. This

constraint is expressed as a denial constraint as follows (we use new identifiers for each new denial constraint as they now refer to the relation *hours*):

$$\varphi_1 : \neg(t_x.\text{Role} = t_y.\text{Role} \wedge t_x.\text{Hours} > t_y.\text{Hours} \wedge t_x.\text{Bonus} < t_y.\text{Bonus})$$

In Table 5.1, tuples t_1 and t_2 share the same value of Role. Between those two, tuple t_1 has the highest value of Hours, so it should not have the lowest value of Bonus. This means that the pair of tuples (t_1, t_2) is a violation of φ_1, and hence Table 5.1 is inconsistent.

Table 5.1: An instance of the relation *hours*.

	EmpID	ProjID	Role	Hours	Bonus
t_1	E1	P1	Developer	4	$2000
t_2	E2	P1	Developer	2	$3000
t_3	E3	P1	Developer	4	$4000
t_4	E1	P2	DBA	4	$4000

5.1.2 Detection of denial constraint violations

A naive approach to detect the violations of a denial constraint is to evaluate its conjunction of predicates for each pair of tuples. If the evaluation is true, then we add that pair of tuples to the result. This approach exhibits a quadratic time complexity in the number of records, which can be computationally prohibitive for large relations. A straightforward alternative is to use SQL with the query processing capabilities of database management systems. However, this might not eliminate the quadratic complexity either, as we discuss next.

The predicates of denial constraints compare the values of columns between two tuples of the same table. Therefore, a simple self-join query using the predicates of the denial constraint in the where clause exposes the violations. The following example shows a SQL query that finds the EmpID's of tuple pairs that violate the denial constraint φ_1:

```
1  select tx.EmpID, ty.EmpID
2  from   hours tx, hours ty
3  where  tx.Role = ty.Role
4  and    tx.Hours > ty.Hours
5  and    tx.Bonus < ty.Bonus;
```

Related work has reported that self-joins (and mainly inequality self-joins) have received little attention in commercial database management systems [141]. Indeed, our experiments with three different database management systems exposed two main issues: (i) excessive memory requirements; and (ii) use of ineffective join algorithms. Some database management systems

run out of memory or took more than one hour to execute queries for common functional dependencies on samples with $200K$ tuples. In addition, most database management systems rely on nested-loop approaches for self-joins with range predicates, which may result in extremely long runtimes.

Indices might not help either: the conditions to detect violations often require validating all the records with table scans. The database management system may not use the indices in the query plans, and the few cases that indices are chosen do not pay off for the costs of index creation. One of the reasons for the poor performance of database management systems is the expected cost to materialize self-joins, which is quadratic in the number of records in the worst case [142]. This cost is evident when denial constraints require high-cardinality predicates, such as a range predicate for an order dependency with many qualifying tuples.

5.1.3 Previous solutions for detection of denial constraint violations

Most of the recently presented data cleaning tools use traditional database management systems as their mechanism for detection of denial constraint violations [35, 105, 34]. These tools inherit the performance issues discussed earlier, and their evaluation experiments use only small datasets or only simple dependencies, such as functional dependencies. Implementing a dedicated denial constraint violation module is an alternative, for instance, Chu et al. do so using pairwise comparisons [54]. However, their experimental evaluation also uses only a small number of records (i.e., up to $100K$ tuples).

Closer to our work is the denial constraint violation detection component of HYDRA – a state-of-the-art algorithm for denial constraint discovery [55]. Efficient detection of denial constraint violations is a central part of the algorithm, so the authors have proposed novel techniques to handle the problem. There are two main ideas in this component: The use of specialized data structures; and the customization of algorithms for different predicate types. While these ideas have inspired our project, the way VIOFINDER organizes and operates on its data structures is different from HYDRA. For example, HYDRA uses the IEJOIN algorithm, which has been shown to deliver efficient performance for self-joins based on range predicates [141]. Our system, in turn, uses a novel sort-merge approach that can be even faster than IEJOIN. We also use different approaches for other types of predicate, as discussed later in this chapter. We use HYDRA and IEJOIN as the main baselines in our experimental evaluation.

5.2 THE VIOFINDER SYSTEM

VIOFINDER is designed to deliver robust performance for different types of data dependencies. In this section, we introduce key ideas that enable VIOFINDER to avoid the issues outlined in Section 5.1.2, e.g., nested-loop joins and materialization overhead. We describe specialized data structures in Sections 5.2.1 and 5.2.3; key operations in 5.2.2; and the architecture of VIOFINDER in Section 5.2.4.

5.2.1 Cluster, cluster pairs, and partitions

We use specialized data structures to represent enumerations of pairs of tuples in a compact manner. A *cluster* c is a set of tuple identifiers (the tuple position within the table). A *cluster pair* is an ordered pair (c_1, c_2) that represents the set of all pairs of tuples $(\mathsf{t}_x, \mathsf{t}_y)$, such that $\mathsf{t}_x \in c_1$, $\mathsf{t}_y \in c_2$ and $\mathsf{t}_x \neq \mathsf{t}_y$. For instance, the cluster pair $(\{\mathsf{t}_1\}, \{\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3\})$ represents the set of pairs of tuples $(\mathsf{t}_1, \mathsf{t}_2), (\mathsf{t}_1, \mathsf{t}_3)$. A *partition* L is any set of cluster pairs.

Clearly, partitions consume much less memory than exhaustive enumerations of pairs of tuples. For a relation r with n tuples, the cluster pair $(\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}, \{\mathsf{t}_1, \ldots, \mathsf{t}_n\})$ represents the whole Cartesian product $\mathsf{r} \times \mathsf{r}$ using only $2n$ integers, whereas the equivalent enumeration of pairs of tuples requires $n(n-1)$ pairs of integers to do so.

5.2.2 Refinement of columns and partitions

The first key operation of VIOFINDER is the *refinement of columns*. A *column refiner* takes as input one predicate and returns partitions containing cluster pairs that represent every pair of tuples that is true for the input predicate. As an example, consider the refinement of columns for the predicate $\mathsf{t}_x.\mathsf{Role} = \mathsf{t}_y.\mathsf{Role}$ and the records in Table 5.1. The refinement gives us a partition with a single cluster pair: $[(\{\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3\}, \{\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3\})]$ – the cluster pair $(\{\mathsf{t}_4\}, \{\mathsf{t}_4\})$ is discarded since it does not produce any pair of distinct tuples. The main primitive here is a full table scan for each column of the predicate. How to use these scans depends on the type of comparison operator in each predicate. In Section 5.3, we describe how to implement the refinement of columns for the different comparison operators. For now, we assume column refiners to be "black-boxes". Besides, we assume a random sequence of refinements— we discuss how to order refinements for better performance in Section 5.2.5.

The second key operation of VIOFINDER is the *refinement of partitions*. Each *partition refiner* takes as input a predicate and a partition and produces new partitions containing cluster pairs with every pair of tuples that is true for the input predicate, and of course, true for the predicates in the past refinements that produced the input partition. As an example, consider again the partition from predicate $\mathsf{t}_x.\mathsf{Role} = \mathsf{t}_y.\mathsf{Role}$ described earlier: $[(\{\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3\}, \{\mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3\})]$. Pushing this partition into the refinement of partitions for the predicate $\mathsf{t}_x.\mathsf{Hours} > \mathsf{t}_y.\mathsf{Hours}$ produces the partition: $[(\{\mathsf{t}_1, \mathsf{t}_3\}, \{\mathsf{t}_2\})]$. If we push this last partition further into the refinement of partitions for the predicate $\mathsf{t}_x.\mathsf{Bonus} < \mathsf{t}_y.\mathsf{Bonus}$, we obtain the partition $[(\{\mathsf{t}_1\}, \{\mathsf{t}_2\})]$. This partition represents the violations of the denial constraint φ_1. The refinement of partitions is similar to the refinement of columns. However, the former requires fetching only the values of columns of the tuples in the partitions, instead of entire columns as the latter requires. Another difference is in the type of optimizations we can use in each type of refinement, which are described in Section 5.3.

5.2.3 Cluster indexes

A common step in the refinement of columns is the creation of cluster indexes on the columns of predicates. Let V be the set of values in the domain of column A. For every value $v \in V$, we assign a cluster c with all tuples having v as the value in column A. The cluster index $\mathscr{H}_{\mathsf{A}}$ is a hash map where each entry maps a value $v \in V$ into its cluster c. For instance, the cluster index $\mathscr{H}_{\mathsf{Role}}$ is: $[\langle$"Developer", $\{t_1, t_2, t_3\}\rangle, \langle$"DBA", $\{t_4\}\rangle]$. Similarly, the refinement of partitions requires the creation of conditioned cluster indexes $\mathscr{H}_{\mathsf{A},c}$. We fetch column values of the tuples in the cluster then create a hash map such that each distinct value fetched is mapped into a cluster with all tuples having that value. For example, the conditioned cluster index $\mathscr{H}_{\mathsf{Hours},\{t_1,t_2,t_3\}}$ is: $[\langle 2, \{t_2\}\rangle, \langle 4, \{t_1, t_3\}\rangle]$.

We considered three facts to choose an implementation for clusters, which are essentially sets of integers. First, the size of cluster indexes grows linearly with the number of distinct values of a column since these values are mapped to one cluster each. Second, refinement algorithms produce partitions containing many cluster pairs. Third, these algorithms have to compute unions or differences of clusters. These facts led us to employ Roaring (compressed) bitmaps, a hybrid data structure that combines bitmaps with sorted arrays to achieve good compression rates [143]. As a result, we can store large numbers of clusters with many integers using less memory. Besides, Roaring bitmaps perform fast unions and differences as bitwise `OR` and `AND NOT` operations which are, in many cases, even faster than non-compressed counterparts. For algorithmic details on Roaring bitmaps, we refer the reader to [143].

5.2.4 System overview

VIOFINDER assigns a refiner to each denial constraint predicate, based on the predicate's form, and refiners connect with each other through a partition pipeline. Each column of the dataset is kept as an in-memory array so that refiners can fetch the values of the columns in their predicates. Partition pipelines work as push-based iterations. Figure 5.1 illustrates a pipeline with three refiners. Each partition is linked to either a next refiner or to the output. In the former case, the current refiner produces a new partition and pushes it to the next refiner, which immediately starts consuming the cluster pairs one by one. In the latter case, no more refinement is necessary, so partitions are pushed to the output. At this point, the concrete violations are materialized.

The partition pipeline has the following properties:

Customizable refinement. Conceptually, refiners implement a produce/consume interface so that different refinement implementations and optimizations can be used at different stages of the pipeline. Instead of using a general-purpose refinement strategy (e.g., nested loop), VIOFINDER uses different refinement strategies depending on the form of the predicate.

Controlled intermediates. Some refinements might produce large intermediates. To avoid excessive resource utilization, our refinement algorithms check the size of current partitions

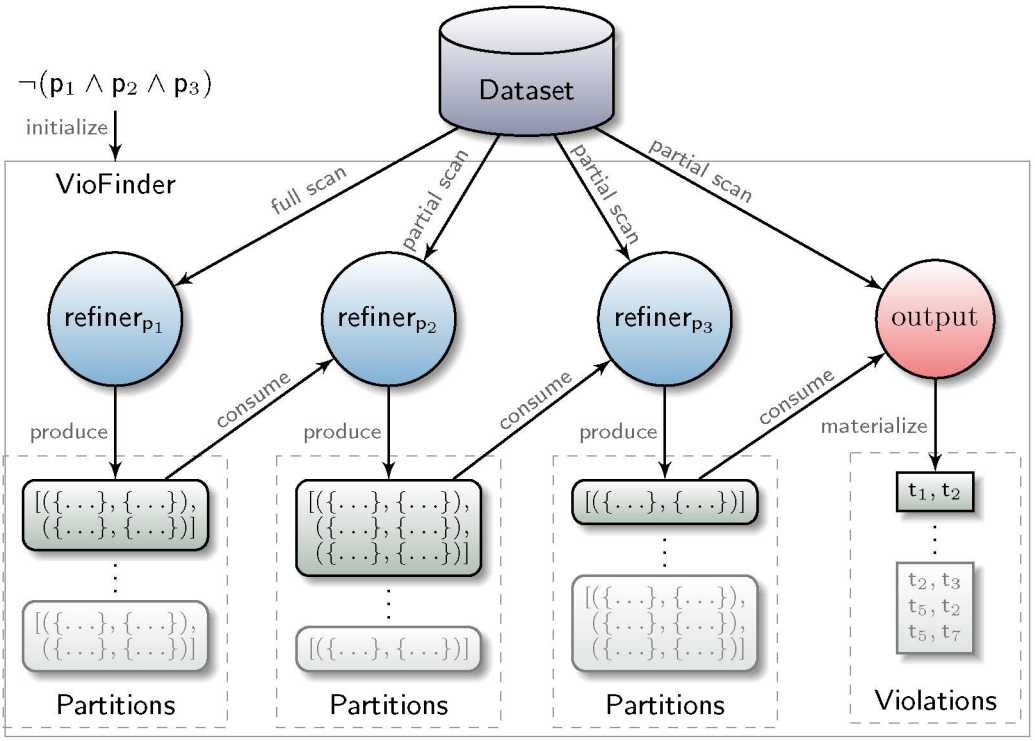


Figure 5.1: Example of a partition pipeline.

before pushing new tuples into the pipeline. As a result, refinements can use logical optimizations that work for multiple tuples at a time, while avoiding materializing large intermediates.

Late materialization. VIOFINDER does not fully materialize tuples until after the last refinement in the pipeline has been processed. As a result, refiners need to fetch only the values of the columns of its predicates—partition refiners in particular do so only for tuples from previous refinements. Such a scheme maximizes the use of memory bandwidth: only the necessary parts of relevant tuples are fetched in each stage of the pipeline.

Cluster pair processing. The actual refinement is computed at the level of cluster pairs with four primary steps:

1. Iteration over the tuples in each cluster—a tight loop suffices to iterate entire clusters fast because they usually have far fewer tuples than the relation.

2. Fetch of column values—as already mentioned, only the column values that are relevant for a refinement are fetched.

3. Build of auxiliary data structures—the auxiliary data structures in refinements usually have a low memory footprint since they grow with the clusters.

4. Refinement logic—some forms of partitions allow refinements to skip tuple fetches, which improves performance.

These properties also apply to column refiners, with the difference that entire columns are fetched in Steps 1 and 2.

5.2.5 Order of refinements

The *order* of the denial constraint predicates (and, therefore, refinements) has a significant impact on performance. Choosing a poor predicate order might produce very large intermediate partitions, causing significant overhead in intermediate refinements. We choose the order of predicates based on predicate selectivity. The selectivity of a predicate is the fraction of pairs of tuples in a relation that satisfy that predicate. We estimate approximate selectivities using a small random sample of pairs of tuples (without replacement), then we order the predicates from most selective to least selective. This technique is also used in HYDRA [55]; however, the algorithm uses a larger sample. For every tuple in the dataset, HYDRA samples a small number of other tuples to form pair of tuples. We found that using a fixed small sample bounded to $1M$ elements produce the same predicate order, and it is faster to estimate. We refer to [144] for a deeper discussion on selectivity estimation; such a discussion is beyond the scope of this thesis.

5.3 REFINEMENT ALGORITHMS

Denial constraints support predicates of several different forms for backing a wide range of data dependencies. These predicates include comparison using different operators within a single column or across two different columns. In this section, we present refinement algorithms that take the predicate form into account for efficiency. For convenience, we divide the presentation of these algorithms into equijoins with the *equal to* operator $\{=\}$, antijoins with the *not equal to* operator $\{\neq\}$, and non-equijoins with *range* operators $\{<, \leq, >, \geq\}$. Most of the algorithms operate for a single predicate at a time. There is one particular case in which the refinement combines multiple predicates for better performance.

5.3.1 Equijoins

The most basic form of refinement is the refinement of columns for equality predicates on a single column, such as $\mathsf{t}_x.\mathsf{A} = \mathsf{t}_y.\mathsf{A}$. The first step is to build a cluster index $\mathscr{H}_\mathsf{A}$. Each cluster c of $\mathscr{H}_\mathsf{A}$ is precisely a set of tuples having the same value v, so we can use cluster pairs in the form of reflexive relations (c, c) to represent all pairs of tuples that have the same value v in column A. Clusters with only one tuple are ignored, because they cannot produce pairs of distinct tuples. We insert each valid cluster pair (c, c) into the output partition L and check its size. If the number of cluster pairs in the partition L exceeds a threshold, we stop iterating the clusters in the cluster index $\mathscr{H}_\mathsf{A}$, and push the partition L into the next level of the pipeline. Of course, the next call to the refiner skips the clusters pairs of $\mathscr{H}_\mathsf{A}$ that have already been processed.

Some refinements might produce large intermediate results. After all, refinements are equivalent to the self-joins in the predicates of a denial constraint, which often join non-key columns. Nonetheless, we can avoid the full materialization of large intermediates by controlling the size of partitions currently being processed, as in our first refinement algorithm. Refinements

stop producing new cluster pairs as soon as the number of cluster pairs in a partition exceeds a threshold, or when it has no more pairs of tuples to compute. In the former case, the state of the refinement is saved so that a next call to it starts producing new cluster pairs from where it stopped earlier. For simplicity, we do not elaborate on these procedures for the remainder refinement algorithms.

Equality predicates on single columns are very common in denial constraints. For instance, denial constraints use them to represent unique constraints or the left hand side of functional dependencies. Refinements of this type of predicate are related to a concept in dependency discovery known as equivalence classes [22]. Compared to other forms of predicates, equality predicates have lower selectivity so that ordering the refinements put them first in the pipeline. In addition, we observe that partitions can only reduce in size as they go through the pipeline stages for sets of predicates with this form. For instance, the partition for predicate $\mathsf{t}_x.\mathsf{Role} = \mathsf{t}_y.\mathsf{Role}$ is $[(\{\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3\},\{\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3\})]$, and the partition for the conjunction of predicates $\mathsf{t}_x.\mathsf{Role} = \mathsf{t}_y.\mathsf{Role} \wedge \mathsf{t}_x.\mathsf{Hours} = \mathsf{t}_y.\mathsf{Hours}$ is $[(\{\mathsf{t}_1,\mathsf{t}_3\},\{\mathsf{t}_1,\mathsf{t}_3\})]$. We take advantage of this fact with a code pattern that reduces clusters as fast as possible and, therefore, reduces the materialization of intermediate partitions.

Algorithm 4 is a special case of refinement that handles multiple predicates at once, namely multiple equality predicates on single columns. In the initial call `refineCluster`(c_r, A_1, L) in Line 12 we build a cluster index with every tuple in the table. When we call `refineCluster`(c, A_i, L) for $i > 1$, every tuple in c have the same combination of values in columns $\mathsf{A}_1,\ldots,\mathsf{A}_{i-1}$. As a consequence, the tuples in the clusters c' of the conditioned cluster index $\mathscr{H}_{\mathsf{A}_i,c}$ (Line 2) have the same combination of values in columns $\mathsf{A}_1,\ldots,\mathsf{A}_i$. The base case occurs when there are no further predicates to check, in which case we insert cluster pair (c,c) into the output partition L (Line 9).

Algorithm 4: Refinement of columns for predicate sequence of the form $\mathsf{p}_1\colon \mathsf{t}_x.\mathsf{A}_1 = \mathsf{t}_y.\mathsf{A}_1,\ldots,\mathsf{p}_i\colon \mathsf{t}_x.\mathsf{A}_\mathsf{i} = \mathsf{t}_y.\mathsf{A}_\mathsf{i}$

```
1  Function refineCluster(c, A_i, L)
2      let H_{A_i,c} be a conditioned cluster index
3      let C' be the set of clusters in H_{A_i,c}
4      foreach c' ∈ C' do
5          if c'.size > 1 then
6              if there exists a predicate p_{i+1} then
7                  refineCluster(c', A_{i+1}, L)
8              else
9                  Insert cluster pair (c', c') into L
10 initialize an empty partition L
11 initialize a cluster c_r with every tuple of table r
12 refineCluster(c_r, A_1, L)
13 return L
```

The refinement of columns for equality predicates on two *different* columns, $\mathsf{t}_x.\mathsf{A} = \mathsf{t}_y.\mathsf{B}$, is similar to traditional hash joins. We first build cluster indexes $\mathscr{H}_\mathsf{A}$ and $\mathscr{H}_\mathsf{B}$. The cluster index $\mathscr{H}_\mathsf{A}$ acts as the "build input", whereas cluster index $\mathscr{H}_\mathsf{B}$ acts as the "probe input"—we assume column A to produce fewer entries than column B. We iterate the values v in the cluster index $\mathscr{H}_\mathsf{A}$ and, for each of those, we probe cluster index $\mathscr{H}_\mathsf{B}$. If cluster index $\mathscr{H}_\mathsf{B}$ contains the value v, then we combine the cluster assigned to the value v in $\mathscr{H}_\mathsf{A}$, denoted c_A, with the cluster assigned to the value v in $\mathscr{H}_\mathsf{B}$, denoted c_B. The cluster pair $(c_\mathsf{A}, c_\mathsf{B})$ indicates that every tuple $\mathsf{t} \in \mathsf{c}_\mathsf{A}$ have the same value in column A, which is equal to the value of every tuple $\mathsf{t}_y \in \mathsf{c}_\mathsf{B}$ in column B.

The refinement of partitions for an equality predicate on two (not necessarily different) columns is shown in Algorithm 5. We iterate each cluster pair (c_1, c_2) in the input partition, for which we retrieve two conditioned cluster indexes: $\mathscr{H}_{\mathsf{A},c_1}$ and $\mathscr{H}_{\mathsf{B},c_2}$. The remainder of the algorithm is analogous to the refinement of columns for equality predicates on two different columns. The difference is that build-inputs are conditioned cluster indexes $\mathscr{H}_{\mathsf{A},c_1}$, whereas probe-inputs are conditioned cluster indexes $\mathscr{H}_{\mathsf{B},c_2}$.

Algorithm 5: Refinement of partition L_{in} for predicates of the form $\mathsf{t}_x.\mathsf{A} = \mathsf{t}_y.\mathsf{B}$ (A and B can be equal)

```
1  initialize an empty partition L_out
2  foreach cluster pair (c1, c2) ∈ L_in do
3  |  let H_{A,c1} and H_{B,c2} be conditioned cluster indexes
4  |  let V be the set of values in H_{A,c1}
5  |  foreach v ∈ V do
6  |  |  c_B ← H_B(v)
7  |  |  if c_B is not null then
8  |  |  |  c_A ← H_A(v)
9  |  |  |  Insert cluster pair (c_A, c_B) into L_out
10 return L_out
```

The algorithms we have presented so far take linear time in the number of tuples. In short, we fetch column values, build cluster indexes using hashing, and iterate the entries in these clusters to emit partitions.

5.3.2 Antijoins

The following refinement of columns detects pairs of tuples having different values of a single column, i.e., predicates of the form $\mathsf{t}_x.\mathsf{A} \neq \mathsf{t}_y.\mathsf{A}$. We need to insert cluster pairs (c, c') into the result partition: Cluster c is each cluster of the cluster index $\mathscr{H}_\mathsf{A}$; and cluster c' is the relative complement of cluster c in a cluster with all tuples in the table c_r— also termed set difference $c' \leftarrow c_\mathsf{r} \setminus c$. We do as follows to detect pairs of tuples having different values for two different columns, $\mathsf{t}_x.\mathsf{A} \neq \mathsf{t}_y.\mathsf{B}$. Given an entry $\langle v, c_\mathsf{A} \rangle$ in the cluster index $\mathscr{H}_\mathsf{A}$, we check whether there exists an entry $\langle v, c_\mathsf{B} \rangle$ in the cluster index $\mathscr{H}_\mathsf{B}$. If so, we insert a cluster pair (c_A, c') into the result partition, where $c' \leftarrow c_\mathsf{r} \setminus c_\mathsf{B}$. Otherwise, the value in column A of the tuples in c_A is

different from the values in column B of every tuple in the table; then we insert a cluster (c_A, c_r) into the result.

The refinement of partitions for antijoin predicates is given as Algorithm 6. For each cluster pair (c_1, c_2) in the the input partition L_{in}, we retrieve the conditioned cluster indexes $\mathscr{H}_{A,c_1}$ and $\mathscr{H}_{B,c_2}$. Then, for each value v (with assigned cluster c_A) of cluster index $\mathscr{H}_{A,c_1}$ we search for a cluster c_B in the cluster index $\mathscr{H}_{B,c_2}$. In a successful search, we use the relative complement of cluster c_B in the cluster c_2 to form the result cluster pair with c_A (Lines 8–10). Otherwise, cluster c_2 has no tuple whose value in B is v, so it can be directly combined with cluster c_A in Line 12.

Algorithm 6: Refinement of partition L_{in} for predicates of the form $t_x.A \neq t_y.B$ (A and B can be equal)

```
1  initialize an empty partition L_out
2  foreach cluster pair (c1, c2) ∈ L_in do
3  |  let H_{A,c1} and H_{B,c2} be conditioned cluster indexes
4  |  let V be the set of values in H_{A,c1}
5  |  foreach v ∈ V do
6  |  |  c_A ← H_A(v)
7  |  |  c_B ← H_B(v)
8  |  |  if c_B is not null then
9  |  |  |  c' ← c2 \ c_B
10 |  |  |  Insert cluster pair (c_A, c') into L_out
11 |  |  else
12 |  |  |  Insert cluster pair (c_A, c2) into L_out
13 return L_out
```

With regard to time complexity, building and probing cluster indexes takes linear time in the number of tuples. In addition, the algorithms for antijoin predicates have the additional cost of set difference operations (e.g., Line 9 in Algorithm 6). The selectivities of these types of predicates are usually high, so their respective refinements might produce large intermediate partitions. In practice, these type of refinement come last in the pipeline, at a point where most pair of tuples have already been filtered out.

5.3.3 Non-equijoins with range operators

Let us next consider the refinement of columns for range predicates of the form $t_x.A > t_y.A$. We build the cluster index $\mathscr{H}_A$ and sort its entries in ascending order according to the keys (the distinct values of the column). For clarity, we denote such sorted maps with $\vec{\mathscr{H}}_A$. For the sorted entries $\langle v_1, c_1\rangle, \ldots, \langle v_i, c_i\rangle \in \vec{\mathscr{H}}_A$ we have the following: Every tuple in the cluster c_i has a value v_i that is greater than the values v_j in the tuples of clusters c_j, for all $j < i$. For each cluster c_i, we form a cluster pair (c_i, c_i') such that $c_i' = \bigcup_{j=1}^{i-1} c_j$. At each iteration i, we compute the cluster c_i' using a copy of the last cluster c_{i-1}' and only one union operation. Finally, we insert

each cluster pair $(c_i, c_i{}')$ into the output partition L. For predicates of the form $\mathsf{t}_x.\mathsf{A} >= \mathsf{t}_y.\mathsf{A}$ we must include c_i into the right hand side of the cluster pair, so we compute clusters $c_i{}' = \bigcup_{j=1}^{i} c_j$. The algorithm is symmetric for predicates of the form $\mathsf{t}_x.\mathsf{A} < \mathsf{t}_y.\mathsf{A}$ and $\mathsf{t}_x.\mathsf{A} \leq \mathsf{t}_y.\mathsf{A}$ with the entries of the cluster index $\mathscr{H}_\mathsf{A}$ in descending order according to the keys. In the worst case, the values of column A are all distinct, thus, cluster indexes have n entries. In this case, the time complexity is dominated by the time spent to sort these n entries plus the time to perform n union operations.

The remaining of the refinement algorithms are based on the sort-merge paradigm. The general idea is to iterate sorted cluster indexes to incrementally find and build matching cluster pairs from previous iterations. Algorithm 7 shows the refinement of columns for a predicate on two different columns, such as $\mathsf{t}_x.\mathsf{A} > \mathsf{t}_y.\mathsf{B}$. After building sorted cluster indexes $\vec{\mathscr{H}}_\mathsf{A}$ and $\vec{\mathscr{H}}_\mathsf{B}$ in Line 2, we filter their values out for those entries that cannot form cluster pairs that satisfy the predicate. That is, we remove from cluster index $\vec{\mathscr{H}}_\mathsf{A}$ the entries with values that are smaller than the smallest value of cluster index $\vec{\mathscr{H}}_\mathsf{B}$, and from $\vec{\mathscr{H}}_\mathsf{B}$ the entries with values that are greater than the greatest value of $\vec{\mathscr{H}}_\mathsf{A}$. If the cluster indexes $\vec{\mathscr{H}}_\mathsf{A}$ and $\vec{\mathscr{H}}_\mathsf{B}$ are empty at this point, there are no matching cluster pairs so the algorithm returns an empty partition. Otherwise, the first entry $\langle v_\mathsf{high}, c_\mathsf{high}\rangle$ of cluster index $\vec{\mathscr{H}}_\mathsf{A}$ has a value that is strictly greater than the value of the first entry $\langle v_\mathsf{low}, c_\mathsf{low}\rangle$ of cluster index $\vec{\mathscr{H}}_\mathsf{B}$, so we form the first cluster pair that satisfy the predicate (Lines 4– 7). Such cluster pairs are kept in variables *pair* that are updated as we find new matching clusters.

The merge part of the algorithm begins in Line 9. We use the value v_high used to form the current *pair* and find matching entries $\langle v_\mathsf{low}, c_\mathsf{low}\rangle$ in the cluster index $\vec{\mathscr{H}}_\mathsf{B}$ that also satisfy the predicate. Then, we update the right hand side of *pair* to include the tuples of clusters c_low (Lines 9-12). Whenever we find a non-matching entry, we update the left hand side of *pair* (Lines 14–17). That is because there might be entries in $\vec{\mathscr{H}}_\mathsf{A}$ with values v_high that, despite being smaller than the current v_low, are greater than the values v_low previously used in Lines 9-12. By doing this, we keep as much tuples as possible within the the same cluster pair. We find the starting point of a new matching cluster pair whenever we find a new entry $\langle v_\mathsf{high}, c_\mathsf{high}\rangle$ in $\vec{\mathscr{H}}_\mathsf{A}$ with a value v_high greater than the current v_low (the else clause in Line 18). At this point, the left hand side of the new cluster pair is c_high and its right hand side is the union of the tuples in the current c_low with all tuples in the c_low from previous iterations. In other words, the right hand side of *pair* can only expand. We repeat the while loop in Line 9 until there is no entry in $\vec{\mathscr{H}}_\mathsf{B}$ to visit. Finally, we perform a last update in the left hand side of the last *pair* with any left entry of cluster index $\vec{\mathscr{H}}_\mathsf{A}$ (Lines 24–26).

The time complexity for Algorithm 7 is given by the time spent to build and sort cluster indexes, plus the time spent in merging these clusters. While the merge loop runs in $\mathscr{O}(2n)$ (assuming n entries in each cluster index), performing cluster unions and copies depends on the internal states of their bitmaps.

Algorithm 7: Refinement of columns for predicate of the form $\mathsf{t}_x.\mathsf{A} > \mathsf{t}_y.\mathsf{B}$

```
1  initialize an empty partition L
2  let H⃗_A and H⃗_B be sorted cluster indexes
3  remove from H⃗_A and H⃗_B those entries that do not produce cluster pairs
     for t_x.A > t_y.B
4  ⟨v_high, c_high⟩ ← H⃗_A.next()
5  ⟨v_low, c_low⟩ ← H⃗_B.next()
6  pair ← (c_high, c_low)
7  Insert pair into L
8  if H⃗_A.hasNext() or H⃗_B.hasNext() then
9      while H⃗_B.hasNext() do
10         ⟨v_low, c_low⟩ ← H⃗_B.next()
11         if v_high > v_low then
12             pair.rhs ← pair.rhs ∪ c_low
13         else
14             while H⃗_A.hasNext() do
15                 ⟨v_high, c_high⟩ ← H⃗_A.next()
16                 if v_high <= v_low then
17                     pair.lhs ← pair.lhs ∪ c_high
18                 else
19                     c_temp ← a copy of pair..rhs
20                     c_low ← c_temp ∪ c_low
21                     pair ← (c_high, c_low)
22                     Insert pair into L
23                     break
24     while H⃗_A.hasNext() do
25         ⟨v_high, c_high⟩ ← H⃗_A.next()
26         pair.lhs ← pair.lhs ∪ c_high
27 return L
```

Algorithm 7 requires minor changes to work with operator $\geq$, and it is symmetric for operators in $\{<, \leq\}$, with cluster indexes $\vec{\mathscr{H}}_{\mathsf{A}}$ and $\vec{\mathscr{H}}_{\mathsf{B}}$ sorted in descending order of keys. The refinement of partitions for predicates with operators in $\{>, \geq, <, \leq\}$ and two (not necessarily different) columns also follows Algorithm 7 with minor changes. The starting point is building conditioned cluster indexes for each cluster pair in the input partition. The remainder of the algorithm is the same as described above.

5.3.4 Cached cluster indexes

The partitions produced by refinements of range predicates, with operators in $\{>, \geq, <, \leq\}$, have a great deal of redundancy across the right hand sides of their cluster pairs. As an example, observe the output of the refinement of columns for predicate $\mathsf{t}_x.\mathsf{Bonus} < \mathsf{t}_y.\mathsf{Bonus}$: $[(\{\mathsf{t}_2\}, \{\mathsf{t}_1\}), (\{\mathsf{t}_3, \mathsf{t}_4\}, \{\mathsf{t}_1, \mathsf{t}_2\})]$. If we were to compute conditioned cluster indexes

for cluster $\{t_1\}$ and $\{t_1, t_2\}$ from scratch, we would require to fetch tuple t_1 twice instead of just once. For larger clusters, the waste would be high, and the running time would increase dramatically. To avoid unnecessary tuple fetches, VIOFINDER employs a simple, but efficient, cache mechanism.

The cache works for the refinement of partitions holding incremental redundancy on the right hand side of their cluster pairs. Such partitions derive from refinements (of both columns or partitions) that use predicates with operators in $\{>, \geq, <, \leq\}$. VIOFINDER maintains a conditioned cluster index $\mathscr{H}_{\mathsf{A},c_{cache}}$, where cluster c_{cache} is a set of tuples that had its values of column A already fetched. Assume we are about to build a conditioned cluster index $\mathscr{H}_{\mathsf{A},c}$. We compute the relative difference of c_{cache} in c, that is, $c_{diff} = c \setminus c_{cache}$. If this result is non-empty, then we already have a portion of the cluster index $\mathscr{H}_{\mathsf{A},c}$ as the cluster index $\mathscr{H}_{\mathsf{A},c_{cache}}$. In this case, we fetch the remaining values of column A we need, that is, the tuples of c_{diff}. We use these values to update $\mathscr{H}_{\mathsf{A},c_{cache}}$. At this point, the cluster index $\mathscr{H}_{\mathsf{A},c_{cache}}$ holds the entries required for $\mathscr{H}_{\mathsf{A},c}$, so we can proceed with the remaining parts of the refinement. On the other hand, an empty result of the relative difference means that the sequence of redundant tuple has stopped, so we can no longer use the previous $\mathscr{H}_{\mathsf{A},c_{cache}}$. In this case, we must build the a new cluster index $\mathscr{H}_{\mathsf{A},c_{cache}}$ from scratch.

5.4 EXPERIMENTAL EVALUATION

We ran several experiments with VIOFINDER, three database management systems, and a system tailored for denial constraints. In this section, we compare the performance of these systems and analyze the design choices of VIOFINDER.

5.4.1 Experimental setup

Datasets and denial constraints. We used three datasets and eight denial constraints, as shown in Table 5.2. The `Tax` dataset is a synthetic compilation of tax-records of US individuals. We generated various `Tax` instances (with up to $100M$ records) using the data generator from [24]. The denial constraints φ_3–φ_5 are defined for the single table of the `Tax` dataset. The `TPC-H` dataset is extracted from the synthetic TPC-H benchmark. We used a scale factor of ten to produce `TPC-H` instances with up to $60M$ records. The denial constraint φ_6 is defined for the denormalization of tables *lineitem* and *orders*, and the denial constraints φ_7 and φ_8 are defined for the *lineitem* table alone. The `IMDB` dataset is extracted from the real-world movie dataset described in [145]. The denial constraint φ_9 is defined for the denormalization (with up to $2.5M$ records) of tables *title* and *kind_type*, and the denial constraint φ_{10} is defined for the denormalization (with up to $5.8M$ records) of tables *cast_info*, *title*, *aka_name*, *name*, *role_type*, and *char_name*. These denial constraints were designed to cover various types of dependencies: Unique constraints (φ_3, and φ_{10}), functional dependencies (φ_2 and φ_9), order dependencies (φ_7), and other dependencies with complex relationships (φ_5, φ_6, and φ_8). Even though some of

them may not hold in production, they have complex predicate structures that challenge the performance of the evaluated systems.

Table 5.2: Datasets and denial constraints for experiments.

Dataset	Denial constraint
Tax	$\varphi_3: \neg(t.\text{AreaCode} = t'.\text{AreaCode} \wedge t.\text{Phone} = t'.\text{Phone})$
Tax	$\varphi_4: \neg(t.\text{State} = t'.\text{State} \wedge t.\text{HasChild} = t'.\text{HasChild} \wedge t.\text{ChildExemp} \neq t'.\text{ChildExemp})$
Tax	$\varphi_5: \neg(t.\text{State} = t'.\text{State} \wedge t.\text{Salary} > t'.\text{Salary} \wedge t.\text{Rate} < t'.\text{Rate})$
TPC-H	$\varphi_6: \neg(t.\text{Customer} = t'.\text{Supplier} \wedge t.\text{Supplier} = t'.\text{Customer})$
TPC-H	$\varphi_7: \neg(t.\text{Extended\_price} > t'.\text{Extended\_price} \wedge t.\text{Discount} < t'.\text{Discount})$
TPC-H	$\varphi_8: \neg(t.\text{Receiptdate} \geq t'.\text{Shipdate} \wedge t.\text{Shipdate} \leq t'.\text{Receiptdate})$
IMDB	$\varphi_9: \neg(t.\text{Title} = t'.\text{Title} \wedge t.\text{ProductionYear} = t'.\text{ProductionYear} \wedge t.\text{Kind} \neq t'.\text{Kind})$
IMDB	$\varphi_{10}: \neg(t.\text{Title} = t'.\text{Title} \wedge t.\text{Role} = t'.\text{Role} \wedge t.\text{Name} = t'.\text{Name} \wedge t.\text{CharName} = t'.\text{CharName})$

Baselines. We compare VIOFINDER with the component for detection of denial constraint violations described in [55], referred to here as HYDRA–IEJOIN. In addition, we compared our system with three database management systems: PostgreSQL (v.12.1), MonetDB (v.11.35.3), and SQLServer (v.2019 CU3). These systems have different query processing models, with different impact on the materialization of intermediate data. PostgreSQL implements the tuple-at-a-time model that moves entire tuples around the memory hierarchy. In contrast, the column-at-a-time processing model of MonetDB fetches only the columns in the SQL statement, but keeps the intermediate data in memory along the entire processing. SQLServer implements a middle ground with a vector-at-a-time model.

Implementation. We implemented VIOFINDER as a standalone tool in Java, that runs in main-memory after dataset loading. We used the Roaring bitmap library to implement clusters [1]. HYDRA–IEJOIN is also a standalone tool that runs in main-memory. We used the Java implementation provided by the authors. To use the database management systems, we translated each denial constraint in Table 5.2 into a SQL query and executed it using the vanilla version of the three database management systems. We created indexes on all predicate columns to investigate if and when the database management systems improve their execution plans. We checked all implementations separately and they all return the same result. We did not need to materialize the violations, so we used a `select count(*)` projection in each query to return only the

[1] https://github.com/RoaringBitmap/RoaringBitmap

number of violations. By the same token, we set the standalone tools to return a count with the number of violations in their output.

Infrastructure and execution. We used a server running Debian 10 (buster) as the experimentation platform. The server is equipped with twelve sockets, each with an Intel(R) Xeon(R) CPU E7-8837 octa-core processor running at 2.67GHz, 756GB of RAM, and 2TB of disk. All executions were single-threaded. VIOFINDER and HYDRA–IEJOIN run on a Oracle's JDK 64-Bit Server VM 1.8.0 with maximum heap size set to 32GB. The numbers in the reports are the average measurement of three independent runs. We used a default threshold of ten cluster pairs for VIOFINDER.

5.4.2 Performance evaluation

Comparison with baselines. We measured the runtime of all denial constraint violation detectors on different datasets and denial constraints. To be able to run the SQL queries within a time limit of 3 hours, we used a sample with $200K$ records of each dataset. Runtimes are broken down into loading, preprocessing, and querying. For the database management systems, these measures are, respectively, the time spent to load the raw files into the database management system, create indexes, and execute the query. For HYDRA-IEJOIN, these measures are, respectively, the time spent to load the raw files into memory, map the input into integer domains plus the time to decide predicate order, and execute the algorithm. VIOFINDER's runtime composition is similar to HYDRA-IEJOIN's, except that it does not include the input mapping time.

Figure 5.2 depicts the measured runtimes of all five systems for all datasets and denial constraints of Table 5.2. In summary, the results in this experiment demonstrate that VIOFINDER performs best in every scenario and that it can be at times orders of magnitude faster than the database management system approaches. For denial constraints φ_7 and φ_8, VIOFINDER finished in a matter of few seconds, PostgreSQL and SQLServer in a matter of few hours, and MonetDB did not finished due to memory limit exceptions. We can see speedups of 1625×, for example, when VIOFINDER is compared to SQLServer for denial constraint φ_8. Moreover, VIOFINDER delivered between 3× and 17.5× faster executions than HYDRA-IEJOIN.

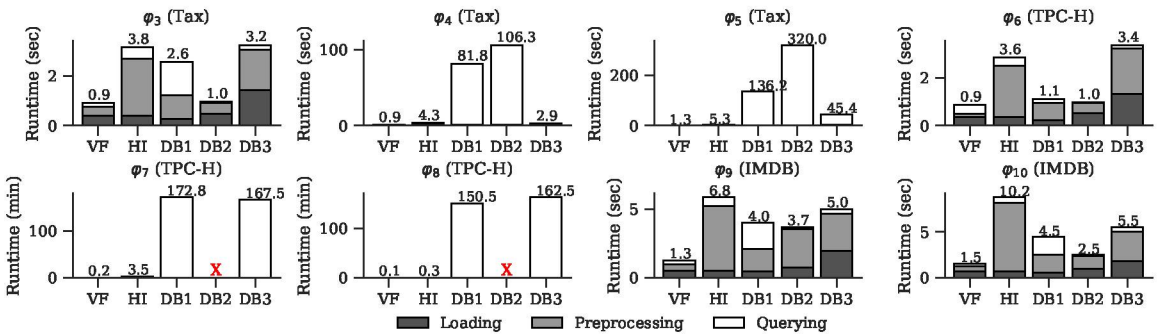


Figure 5.2: Runtime comparison between VIOFINDER (VF), HYDRA-IEJOIN (HI), PostgresSQL (DB1), MonetDB (DB2) and SQLServer (DB3). The datasets are table samples with $200K$ records each.

The execution plans and performance among the evaluated database management systems varied considerably. For the keys in denial constraints φ_3 and φ_{10} and the functional dependency in denial constraint φ_9, PostgreSQL used a Sort-Merge Join approach slower than the HashJoins in SQLServer and MonetDB—we can see the performance impact from algorithm choice in the querying time. All systems used HashJoins for the relationship of mutual inclusion in denial constraint φ_6 and reported fast querying. In contrast, we measured the worst runtimes for denial constraints that express relationships of order between columns (i.e., denial constraints φ_5, φ_7 and φ_8). MonetDB threw memory limit exceptions for denial constraints φ_7 and φ_8 and reported the slowest runtime for denial constraint φ_5. The system used a thetajoin implementation based on Cartesian product that produced large intermediates and impaired performance. PostgreSQL and SQLServer relied on nested loops for those three denial constraints and performed poorly considering the small number of tuples in the experiment. With regards to index usage, the database management systems used table scans for most of the executions due to the selectivity of the predicates. MonetDB and SQLServer used no indices, whereas PostgreSQL used index scans on column Extended_price for the denial constraint φ_7 and on column Shipdate for the denial constraint φ_8. The order of predicate evaluation also influenced performance. For denial constraint φ_4, SQLServer used HashJoins to evaluate the equijoin predicates, then checked the non-equijoin as a residual predicate. The two other database management systems also used HashJoins, but they evaluated the non-equijoin filter first, which yielded in worst runtime. For denial constraint φ_5, all systems evaluated the equijoin predicate first, which helped reducing intermediates and improved performance.

The differences in the executions of the database management systems were expected: after all, they differ from each other internally. These results support our design decisions with VIOFINDER, though. By processing partitions of limited size at-a-time, VIOFINDER bounds the materialization of intermediates. Choosing the order of denial constraint predicates based on predicate selectivity leads VIOFINDER to process predicates that produce smaller intermediates first. In addition, VIOFINDER carefully selects refinement algorithms. Notice that the best results reported by the database management systems uses hash-based approaches. VIOFINDER mirrors this observation and uses hash-like approaches whenever possible. For range predicates, VIOFINDER uses algorithms that are more effective than the nested loop solutions in the database management systems. We observe similar concerns with HYDRA-IEJOIN. However, VIOFINDER spends much less time than HYDRA-IEJOIN in preprocessing.

Scalability in the number of tuples. This experiment considers only querying times (i.e., execution times without loading, index creation, or preprocessing times) because it focuses on the algorithmic efficiency of each system. The previous experiment is a baseline comparison so we used HYDRA-IEJOIN as it was originally conceived by its authors. However, HYDRA-IEJOIN has to map the input into a integer domain, because its implementation is based on integer comparisons. VIOFINDER does not need this step, and also uses a faster approach to

decide predicate order. Thus, to eliminate the additional costs of HYDRA-IEJOIN, we integrated HYDRA-IEJOIN's algorithms into VIOFINDER's platform for this experiment.

Figure 5.3 shows the runtimes (only querying times) measured for the datasets with increasing number of rows—note that some plots have different scales. The plots show SQLServer as the only database management system approach, over only denial constraints without range predicates: None of the database management systems finished execution for denial constraints with range predicates in less than twenty-four hours or without throwing a memory exception. MonetDB faced the same issue executing functional dependencies, and, in the cases PostgresSQL finished, the observed runtimes were orders of magnitude higher than the other database management systems. In practice, SQLServer was the fastest among the database management systems for most denial constraints and datasets. The database management system approach scaled better than VIOFINDER for denial constraint φ_6. The columns in this denial constraint are keys, which database management systems are well-optimized for. In this case, VIOFINDER has less opportunity to use its optimizations (e.g., it does not use Algorithm 4).

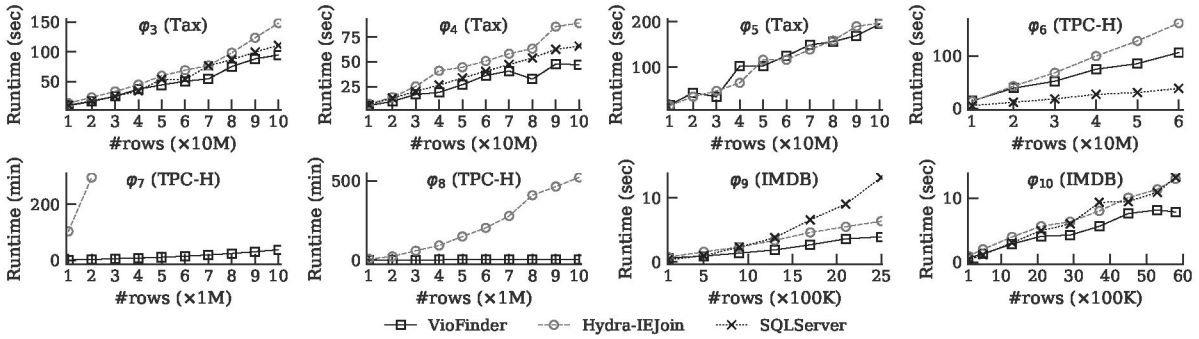


Figure 5.3: Scalability of VIOFINDER, HYDRA-IEJOIN and SQLServer for increasing number of rows.

Although both VIOFINDER and HYDRA-IEJOIN show characteristics of linear growth for denial constraints $\varphi_3, \varphi_4, \varphi_6, \varphi_9$ and φ_{10}, the relative performance difference consistently grows as the number of records grows. Both systems use hash-based approaches with such denial constraints, but differ in key implementation details. VIOFINDER deals with multiple equality predicates on single columns at once (with Algorithm 4), whereas HYDRA-IEJOIN does so one predicate at a time. As a result, HYDRA-IEJOIN requires larger partitions (with larger cluster pairs) to be moved through the pipeline, which may decrease performance. Moreover, VIOFINDER uses bitmaps with sorted arrays to implement set operations (e.g., set difference in different than predicates), whereas HYDRA-IEJOIN uses hash sets. The former approach has been shown to be consistently faster [143].

The performance of VIOFINDER and HYDRA-IEJOIN was roughly similar for denial constraint φ_5, but greatly differed for denial constraints φ_7 and φ_8. For instance, VIOFINDER was on average 307× faster than HYDRA-IEJOIN for denial constraint φ_8 on $10M$ rows. Notice that denial constraints φ_5, φ_7 and φ_8 are those with range predicates. VIOFINDER uses our proposed sort-merge approaches to process range predicates, whereas HYDRA-IEJOIN uses the IEJOIN algorithm [141]. Broadly speaking, both approaches include a phase that builds

auxiliary data structures, and a phase that uses those data structures to produce the results. The costs of the initial phase in the VIOFINDER's sort-merge algorithms consists of building cluster indexes and sorting its entries, and the costs to produce results consists basically of a merge loop that triggers logical operations and copying of bitmaps. In contrast, the IEJOIN algorithm in HYDRA-IEJOIN evaluates two range predicates in a single pass. The initial costs of the algorithm involves computing auxiliary arrays based on sorted versions of column values. As for its second part, the basic idea is to iterate the relative positions of the auxiliary arrays; operate on a bitmap to mark positions of tuples that satisfy the first predicate; then find tuples that also satisfy the second predicate by iterating another auxiliary array and the marked bitmap. The primitives in the second phase of both approaches have a great impact on performance.

We broke down the executions and observed the following. For denial constraint φ_5 the first phase occupied most of the execution time in both approaches, that is, they spent most of the time in sorting. In addition, the refinement of the equality predicate of denial constraint φ_5 occupied only a small fraction of the execution time for both approaches. For denial constraints φ_7 and φ_8, however, both approaches spent most of the time in their second phase. IEJOIN has to iterate auxiliary arrays to find and collect qualifying tuples. For denial constraints with a larger number of violations, as it is the case of denial constraints φ_7 and φ_8, this primitive is heavily penalized because many tuples qualify. In contrast, the sort-merge approach builds the results incrementally from previous iterations with copying of bitmaps. While the approach is also penalized for denial constraints with a large number of violations, its incremental processing saves a great deal of computations and yields lower runtimes.

5.4.3 Additional evaluation of VIOFINDER

The next set of experiments focuses on VIOFINDER. We evaluated the effects that the cache mechanism has on runtime, maximum memory usage, and number of tuple fetches (for refinements that enable caching). We used denial constraint φ_8 because its execution exemplifies how caching can benefit performance. Figure 5.4 shows the measurements using a cache-disabled version of VIOFINDER relative to the measurements using the original—the Y-axis is in log scale. The cache-disabled version has to perform dramatically more tuple fetches and runs considerably slower than its cache-enabled counterpart. The larger the number of tuples in the input, the greater the relative differences in tuple fetches and runtime. Although VIOFINDER consumed more memory using the cache mechanism for fewer tuples (i.e., less than $400K$), it stably consumed about the same amount of memory for larger inputs. This interesting effect happened because the larger inputs produced clusters with higher density that took better advantage of bitmap compression.

Next, we evaluated the impact of varying cluster pair thresholds on runtime and maximum memory usage. We observed that performance and memory usage was relatively stable for small thresholds (i.e., less than 100). Partitions with more than one cluster pair benefited the performance of refinements dealing with a few of tuples at-a-time, because there was less

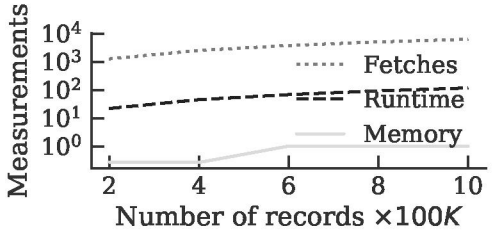


Figure 5.4: Relative impact of caching cluster indexes on denial constraint φ_8.

interpretation overhead. We used a default threshold of 10, because it is the median value of those thresholds that produced the best runtimes for each denial constraint. However, memory usage increased with larger thresholds as partitions are more likely to store more cluster pairs. Large partitions create long-living data objects in the heap that persist for long portions of the pipeline. This effect degrades runtime, because garbage collection needs to perform additional tracing and marking of long-living objects, consuming additional CPU time. Figure 5.5 illustrates such a behavior, for denial constraint φ_8, by showing the memory usage and runtime with increasing tuple pair thresholds relative to these measures with the default threshold.

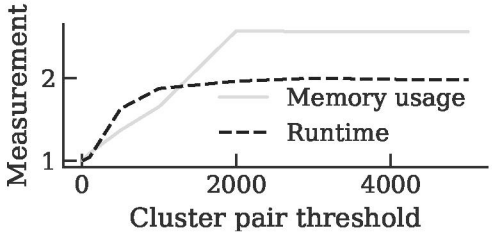


Figure 5.5: Relative impact of increasing cluster pair thresholds on denial constraint φ_8.

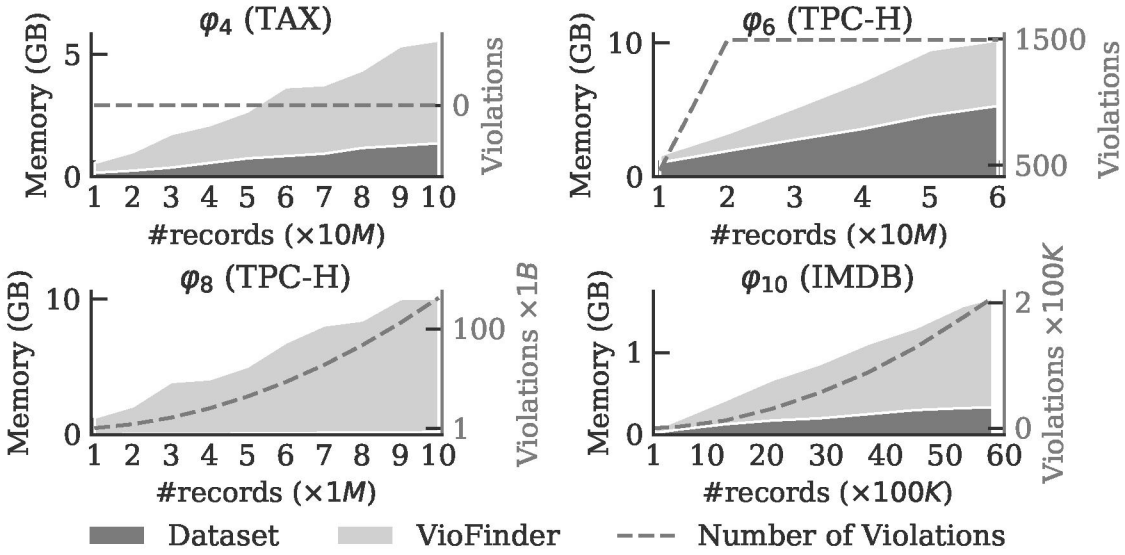


Figure 5.6: Maximum memory usage.

For this last experiment, we measured the size of the in memory data structures storing the datasets, and the maximum memory used by VIOFINDER during each execution. Figure 5.6 shows the results for four denial constraints—the plots include also the number of violations

detected. For most denial constraints, the contributing factor to the linear increase in memory usage is the number of tuples. Notice, however, that denial constraint φ_8 has a huge number of violations. In that case, handling the large intermediates used to produce output consumed much more memory than the in memory datasets. Nonetheless, these results shows that VIOFINDER is not expensive in terms of memory usage.

5.5 SUMMARY

In this chapter, we introduced a system for the detection of denial constraint violations that handle a wide range of data dependencies, from unique constraints to other dependencies that express complex relationships between columns. VIOFINDER shows efficient performance through partition pipelines and effective refinement strategies. Even for larger inputs, or denial constraints that produce sizeable intermediates and results, the performance of our system degrades much more gracefully than the performance of baselines.

Chapter 6

Mind your Dependencies for Semantic Query Optimization

One of the most important data profiling task is dependency discovery, particularly the discovery of the functional dependencies. While functional dependencies are defined as integrity constraints in database design phases, manually updating them as the application and data evolve becomes an error-prone task which may even be left behind in denormalized databases (e.g., data warehouses). In turn, automatic dependency discovery does not rely exclusively on schema information but considers the data tuples of the database as well.

The number of functional dependencies radically increases with the number of columns in the dataset. This number may increase drastically as the number of columns goes up, e.g., in the region of millions for datasets with hundreds of columns and thousands of records [27]. The main problem is that selecting which of the dependencies are most relevant for a given task is left for human analysis. It is particularly challenging to understand the relationships among hundreds, or even thousands, of dependencies spread across multiple relations. Therefore, the selection process should regard the use-case for dependencies. This process should not only prune the unnecessarily large number of results, but it should also provide more meaningfulness to the selected dependencies.

Interestingness measures have been proposed to score functional dependencies and other types of constraints. These measures are primarily based on the statistical properties of the data and have shown good potential to filter dependencies for tasks such as functional dependency evolution [120], data cleansing [137] and normalization [102]. However, those measures may produce inconclusive recommendations to be explored by semantic query optimization [122]. As observed in [146], data dependencies should be exploited with caution. They may impose additional performance penalties in planning phases as the number of dependencies increases.

We present the focused dependency selector (FDSEL), a data-driven, query-aware tool to select relevant functional dependencies for semantic query optimization. We hypothesize that the information from the workload of the application (e.g., selection filters in SQL statements) is a powerful asset to narrow the large number of functional dependencies discovered in the datasets.

First, FDSEL associates summaries of application workloads with the set of discovered functional dependencies in application data. Then, it can use different strategies to recommend sets of functional dependencies that offer the best trade-off between a reduced number of functional dependencies and best gains in query execution time with semantic query optimization. We refer to these sets of functional dependencies as *exemplar functional dependencies*. The FDSEL is also responsible for setting and triggering query optimizations based on query rewritings. The tool acts as a middle-ware between the user applications and the database.

The contributions in this chapter are as follows.

- We present a novel mechanism to combine the semantic information found in functional dependencies and query workload to help in query optimization.
- We formulate effective procedures to select exemplar functional dependencies from the large sets of functional dependencies returned by automatic discovery algorithms.
- We present two schemes in which the exemplar functional dependencies can help in semantic query optimization, namely, join elimination and order optimization.
- We provide an experimental evaluation of our tool, using real and synthetic datasets, which shows that our tool is able to effectively select exemplars with adequate statistical properties, and improve query performance without any human interaction.

The rest of the chapter is organized as follows. Section 6.1 gives an overview of the FDSEL use-case scenario. Section 6.2 details FDSEL. Section 6.3 presents our experimental evaluation of FDSEL. Finally, Section 6.4 concludes the chapter and presents future directions.

6.1 OVERVIEW

In this section, we present a high-level description of FDSEL. Given the high number of functional dependencies discovered in real-world data, the main question we seek to answer is how can we use the semantic information in these dependencies to help in query optimization scenarios effectively. Thus, we design FDSEL as a data-driven, query-aware mediating tool that autonomously leverages semantic query optimizations by exploiting patterns in the data (functional dependencies) and applications (query workload).

Figure 6.1 illustrates the control flow between the components of FDSEL. The input of FDSEL is a database along with its catalog, and a representative query workload. The first component of FDSEL is the *functional dependency extractor*, which uses an efficient algorithm to discover all functional dependencies in the database tables ❶. These functional dependencies determine relationships between groups of attributes and can provide valuable semantic information from the data. FDSEL stores all discovered functional dependencies in a buffer for further analysis.

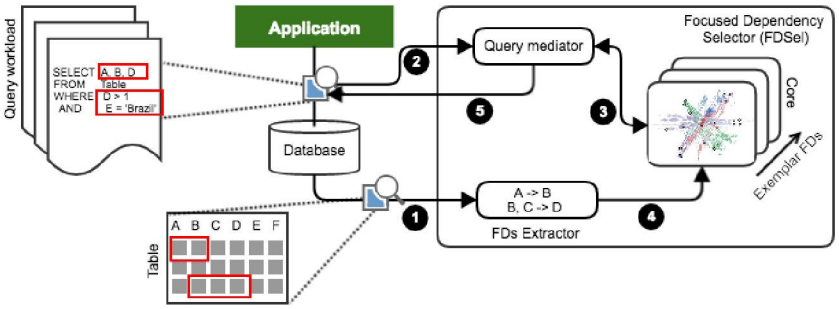


Figure 6.1: FDSEL workflow.

The second component of FDSEL is the *query mediator*, which progressively intercepts the database queries to form a batch ❷. Besides, the query mediator might perform query optimizations for each query, if any optimization is available. These optimizations are set by the *core* component, described later. FDSEL considers the application workload to be lists of queries that are expected to be executed by the application. Once the query mediator has processed sufficient queries, it calls the core component of FDSEL for updates.

The core component receives the workload characterization from the query mediator ❸, scans the functional dependencies buffer, and selects functional dependencies for semantic query optimization ❹. This component counts attribute frequencies from functional dependencies and queries. Then, it combines this frequency information to build a data structure called *occurrence matrices*, which forms the input for the selection procedures. The core component can use three strategies for the selection task. In two of these strategies, we use occurrence matrices to sort functional dependencies. This sorting is based on their structures and their proximity to the query workload. The structure of a functional dependency is defined by which set of attributes define each other, and the proximity of a functional dependency measures how many attributes it has in common with the workload characterization.

The strategies based on ranking are based on two different interest metrics: *distrust*, which considers the redundancy of attribute values, and *Mahalanobis* distance, which considers correlations found in the occurrence matrices. The core component iterates the set of ranked functional dependencies to find functional dependencies with appropriate values for these interest metrics. Finally, the third strategy is an adaptation of the affinity propagation clustering algorithm [147], which works with the occurrence matrices.

The core component is also responsible for setting the rewriting strategies in the query mediator ❺. The component considers only optimizations that preserve semantics; that is, there is no change in the output of the rewritten queries. The FDSEL sits between user applications and data processing platforms, and it is completely decoupled from the internals of any specific database management system.

6.2 FOCUSED DEPENDENCY SELECTOR

In this section, we detail the operation of FDSEL. We describe data structures to combine functional dependencies and workloads, and we present the procedures to select functional dependencies. Finally, we describe how to employ the selected functional dependencies in query optimization.

6.2.1 Discovery of functional dependencies

FDSEL discovers all the non-trivial and minimal functional dependencies holding in the database tables. Several algorithms for functional dependency discovery have been proposed, and many of them have evolved over different versions in the literature. We refer to [21] and [27] for further details on functional dependencies discovery. In practice, FDSEL could use any functional dependency discovery algorithm that, given an instance r, returns the set of non-trivial and minimal functional dependencies over r. FDSEL uses the algorithm HYFD [29], described in Chapter 2. At the time of writing, HYFD was the most efficient functional dependency discovery algorithm as it shows good performance results in terms of runtime and scalability.

6.2.2 Attribute occurrence matrices

Query workloads provide valuable information to support query optimization. In general, a query workload presents strong access patterns, which either in horizontal level (individual tuples) or vertical level (individual attributes), points out to specific database areas that are accessed more frequently than others. In turn, functional dependencies express semantic consistency requirements for data through sets of dependent attributes. FDSEL leverage this characteristic of functional dependencies to reduce the number of sets of attributes that a query optimization should address. Thus, the combination of appropriate semantic information and query workload information is a potential asset to help to find alternative execution strategies and, therefore, improve query processing.

FDSEL measures the binary relationship between a relation's attributes and how often the incoming queries are touching those attributes. This binary relationship is also applied to functional dependencies by only considering their left-hand side and right-hand side attributes. The information about the occurrence of attributes in the queries or functional dependencies is initially stored in a $m \times n$ binary matrix O, called the attribute occurrence matrix (AOM). As a first step, the operations for AOMs regarding queries or functional dependencies are the same; thus, we define all the operations in a single AOM. Throughout the definitions, we distinguish how to adjust each AOM to functional dependencies or queries.

Consider a set of queries $\mathsf{Q} = \{\mathsf{q}_1, ..., \mathsf{q}_m\}$, which we expect to run on the database. For simplicity, we assume there is only one relation in the database. For each query q_i, FDSEL collects the attributes in the operators of q_i. That is, it collects the attributes in the projection

and selection of the query to compose a set of attributes s. Furthermore, for each functional dependency $f : X \rightarrow Y$, FDSEL composes two attribute sets s, one for each side of f.

Consider a relation and its attributes $\mathsf{R}(\mathsf{A}_1, ..., \mathsf{A}_n)$, and a collection of sets $\mathsf{S} = \{\mathsf{s}_1, ..., \mathsf{s}_m\}$ such that $\mathsf{s}_i \subseteq \mathsf{R}$. For each $\mathsf{s}_i \in \mathsf{S}$, and for each $\mathsf{A}_j \in \mathsf{R}$, FDSEL assigns a binary occurrence value for AOM, as in Function 6.1:

$$o_{ij} = \begin{cases} 1 & \text{if } \mathsf{s}_i \text{ has attribute } \mathsf{A}_j, \\ 0 & \text{othewise.} \end{cases} \tag{6.1}$$

Each entry o_{ij} indicates whether or not an attribute of R is touched by one of the elements of s_i.

FDSEL estimates three AOMs. The first one is for a set of queries, denoted by O^{q}. Also, for a given a set of functional dependencies, it estimates an AOM O^{lhs} for their left-hand side; and an AOM O^{rhs} for their right-hand side.

Let $\Sigma = \{\mathsf{A} \rightarrow \mathsf{B}, \mathsf{BC} \rightarrow \mathsf{D}\}$ be the set of functional dependencies discovered in a relation instance r of relation $\mathsf{R}(\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D})$. Besides, consider a set of queries in standard relational algebra $\mathsf{Q} = \{\pi_{(\mathsf{A},\mathsf{D})}(\sigma_{\mathsf{B}=10}(\mathsf{R})); \pi_{(\mathsf{A},\mathsf{B},\mathsf{C})}(\sigma_{\mathsf{C}=20}(\mathsf{R})); \pi_{(\mathsf{A},\mathsf{D})}(\sigma_{\mathsf{D}>1}(\mathsf{R}))\}$. The AOMs O^{q}, O^{lhs}, and O^{rhs} are respectively defined as follows:

$$O^{\mathsf{q}} = \begin{array}{c} \\ \mathsf{q}_1 \\ \mathsf{q}_2 \\ \mathsf{q}_3 \end{array} \begin{array}{c} \begin{array}{cccc} \mathsf{A} & \mathsf{B} & \mathsf{C} & \mathsf{D} \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \end{array},$$

$$O^{lhs} = \begin{array}{c} \\ f_1.lhs \\ f_2.lhs \end{array} \begin{array}{c} \begin{array}{cccc} \mathsf{A} & \mathsf{B} & \mathsf{C} & \mathsf{D} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array},$$

$$O^{rhs} = \begin{array}{c} \\ f_1.rhs \\ f_2.rhs \end{array} \begin{array}{c} \begin{array}{cccc} \mathsf{A} & \mathsf{B} & \mathsf{C} & \mathsf{D} \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array}.$$

The row sum vector of AOM O is given by $\sum_j^m o_{ij}$, and is denoted by $\rho(O)$. Furthermore, let $\gamma(O)$ denote the column sum vector of AOM O, which is given by $\sum_i^n o_{ij}$.

FDSEL requires some additional operations on AOMs O. Notice that each AOM can also be represented as a sequence of rows $O = [o_1, ..., o_m]$. We use a function **elems**(O) that returns the set of elements from O, such that for any o_i and o_k of **elems**(O), then $o_i \neq o_k$. Besides, we use a function **count**(O, o_i) that returns how often an element o_i occurs in the sequence O. Finally, we use a function **length**(o_i) that returns the number of o_{ij} such that $o_{ij} \neq 0$.

Considering two AOMs, O, and O', FDSEL incorporates the number of accesses to the attributes of R of AOM O' into AOM O with a weighted AOM $\mathbf{O}$ given by Equation 6.2.

$$\mathbf{O} = \sum_{i}^{m} O(i)\rho(O') \tag{6.2}$$

It is possible to integrate the attribute weights from the workload into the AOMs of functional dependencies, and vice-versa. However, FDSEL requires only the former type of integration because its goal is to enhance the semantic information of the discovered functional dependencies using workload information.

Consider a weighted AOM $\mathbf{O}$ estimated with Equation 6.2, either $O = O^{lhs}$ or $O = O^{rhs}$, and $O' = O^{\mathsf{q}}$. We use a function **skewed_sort**($\mathbf{O}$) to return a sorted version of **elems**($\mathbf{O}$) that satisfies the following:

$$\text{for any } \mathbf{o}_i \text{ and } \mathbf{o}_{i+1} \text{ from } \mathbf{elems}(\mathbf{O}) \text{ then } \gamma_i(\mathbf{O})\mathbf{count}(\mathbf{O},\mathbf{o}_i) \leq \gamma_{i+1}(\mathbf{O})\mathbf{count}(\mathbf{O},\mathbf{o}_{i+1}). \tag{6.3}$$

The result of **skewed_sort**($\mathbf{O}$) is a sequence of rows sorted according to their frequencies in $\mathbf{O}$ times the weight of their target attributes. Each entry $\mathbf{o}_{ij}$ of **skewed_sort**($\mathbf{O}$) can be converted into the attributes of functional dependencies, left-hand side or right-hand side, by mapping the equivalent attributes A_j of R for each element of $\mathbf{o}_{ij}$ other than zero.

6.2.3 Quality measures for functional dependencies

A variety of quality measures have been proposed to measure dimensions of data dependencies [137, 32, 120]. A standard metric for functional dependencies is redundancy; that is, how often sets of equal values for the left-hand side or right-hand side of functional dependencies jointly appear in the dataset. based on related work [122, 32, 120], We also use data redundancy to measure the distrust of a functional dependency $f : X \rightarrow Y$ in r, as Equation 6.4 shows.

$$d = \sqrt{\left(\frac{|\pi_X(\mathsf{r})|}{|\mathsf{r}|} - \frac{|\pi_Y(\mathsf{r})|}{|\mathsf{r}|}\right)^2} \tag{6.4}$$

The difference in d is minimal when the projections over left-hand side and right-hand side approximate in the number of duplicates. In this case, a functional dependency f is less likely to have been discovered by chance, which reduces the level of distrust of f.

As an example, consider the relation in Table 6.1, and two functional dependencies, $f_1 : \mathsf{AB} \rightarrow \mathsf{C}$ and $f_2 : \mathsf{D} \rightarrow \mathsf{E}$, satisfied by the data. The distrust level of f_1 is given by $d(f_1) = \sqrt{(4/6 - 3/6)^2} = 0.16$, and the distrust level of f_2 is given by $d(f_2) = \sqrt{(5/6 - 1/6)^2} = 0.66$.

Notice that the distrust of a functional dependency f does not consider any workload characteristic. The studies on workload characterization typically investigate many parameters, such as I/O throughput, temporal locality, and data variance aspects. A comprehensive report

Table 6.1: A simple relation.

A	B	C	D	E
b	g	5	1	y
b	g	5	2	y
b	g	5	3	y
b	m	6	4	y
b	q	7	5	y
c	g	7	5	y

on the subject can be seen in [148]. FDSEL uses a second quality measure called *Mahalanobis* distance to combine data instances and workload characterization [149].

Mahalanobis distance works as a similarity measure between the attribute access pattern in the workload and the structure of attributes in the functional dependencies (i.e., left-hand side and right-hand side). We have chosen Mahalanobis distance rather than classical measures, such as Pearson correlation or Euclidean distance because Mahalanobis distance is suitable for side comparisons. For example, Mahalanobis distance agrees with the intuition that "A $\rightarrow$ B is closer to A $\rightarrow$ C" than "A $\rightarrow$ B is to C $\rightarrow$ D", disjointly. The same applies to comparisons between functional dependencies and query workload because they account for the same set of attributes.

Mahalanobis distance uses multi-dimensional analyses of unequal variances and correlations between the weighted attributes of AOMs to adjust the geometrical distribution. Thus, FDSEL can estimate Mahalanobis distances from AOMs. Assume $u = \rho(O^{\mathsf{q}})$ and v to be any $\mathbf{o}_i$ from $\mathbf{O}^{lhs}$, weighted over another O^{q}. FDSEL estimates the Mahalanobis distance as in Equation 6.5:

$$md(u,v) = \sqrt{(u-v)V^{-1}(u-v)^T} \tag{6.5}$$

where V^{-1} is the inverse of the covariance matrix. By using Mahalanobis distances, the difference between query patterns ($\rho(O^{\mathsf{q}})$) and weighted functional dependencies ($\mathbf{O}^{lhs}$) can be considered in terms of the difference between the vectors of u and v relative to their variance.

6.2.4 Selecting functional dependencies

Instead of using all possible rewrite strategies from the large set of functional dependencies, FDSEL uses the properties previously described to focus on meaningful functional dependencies, which we call exemplar functional dependencies. Because FDSEL uses functional dependencies for semantic query optimization, FDSEL focus on exemplars that integrate as much coalescence of attributes as possible while producing the best gains in query optimization. We formulate three different strategies for selecting exemplars, described next.

Selecting functional dependencies based on their rank. FDSEL first sorts the set of discovered functional dependencies Σ using Algorithm 8. The algorithm requires as input a set of functional dependencies, and two weighted AOMs: $\mathbf{O}^{lhs}$ and $\mathbf{O}^{rhs}$. The entries $\mathbf{o}_{ij}$ of

skewed_sort($\mathbf{O}$) are converted into the attributes of functional dependencies by mapping the attributes A_j of R for each element of $\mathbf{o}_{ij}$ other than zero. In other words, the entries $\mathbf{o}_{ij}$ represent valid left-hand side and right-hand side in Σ.

For each distinct left-hand side in Σ, the algorithm iterates each distinct right-hand side to find a combination that builds a valid f in Σ. The combination left-hand side $\rightarrow$ right-hand side is appended to the ranked sequence of functional dependencies Σ' only if such combination is a valid functional dependency in Σ. The iteration over Σ is based on the weighted AOMs with the **skewed_sort** function. The first functional dependencies to be appended in the result Σ' are those in which the target attributes are the most accessed by the application query workload.

Algorithm 8: Ranking functional dependencies

Data: Set of functional dependencies Σ, weighted AOMs $\mathbf{O}^{lhs}$ and $\mathbf{O}^{rhs}$
Result: Ranked functional dependencies Σ'

```
1 Σ' ← { }
2 foreach lhs ∈ skewed_sort(O^lhs) do
3     foreach rhs ∈ skewed_sort(O^rhs) do
4         if (lhs, rhs) build a valid functional dependency f in Σ then
5             f = lhs → rhs
6             Σ' ← {Σ'} + f
```

Algorithm 8 returns the same number of functional dependencies as in the initial set Σ. Because the result Σ' is a sorted sequence, FDSEL can iterate through Σ' until the functional dependencies f in Σ' stop meeting some desired criteria. We noticed that the quality measures of functional dependencies degrade as this iteration occurs. FDSEL estimates the distrust and Mahalanobis distance against the current workload of each functional dependency f, and builds two sets of exemplar functional dependencies. FDSEL outputs the first set of exemplars by considering the following criterion: (1) iterate through Σ' until there is a harsh increase in distrust. The second set of FDSEL is built with the following criterion: (2) iterate through Σ' until there is a harsh increase in Mahalanobis distance against the current query workload. We consider that there is harshness when an element in Σ' shows a quality measure that is higher than the double of the median of previous elements seen in the iteration up to that point. FDSEL extends the set of exemplars using inference rules before applying them in query optimization.

Clustering functional dependencies with affinity propagation algorithm. FDSEL uses clustering in the third strategy to select functional dependencies. To this purpose, we adapted the affinity propagation clustering algorithm to work with AOMs, and cluster functional dependencies based on their weighted structures [147]. Unlike other clustering algorithms (e.g., *k*-means), affinity propagation does not require the number of clusters to be specified a priori. Besides, affinity propagation clustering algorithm can be applied for data that does not lie in a continuous space or data with non-symmetric similarities. The affinity propagation clustering algorithm identifies the most representative elements in a set by recursively transmitting messages between

pairs of elements until convergence. An acceptable set of exemplar functional dependencies (corresponding clusters) is selected when the message-passing procedure is finished. The procedure finishes in two situations: after a fixed number of iterations, or after the message changes fall below a threshold or remain constant for some iterations.

The inputs of the affinity propagation algorithm are measures of similarity between pairs of data points, which FDSEL extracts from the weighted AOMs. Consider two elements $\mathbf{o}_i$ and $\mathbf{o}_j$, $\mathbf{o}_i \neq \mathbf{o}_j$, of AOM $\mathbf{O}^{lhs}$. FDSEL uses Mahalanobis distance as the similarity measure for affinity propagation inputs and estimates the Mahalanobis distance between the left-hand side structures of pairs of functional dependencies. There are two categories of messages exchanged between pairs $[\mathbf{o}_i, \mathbf{o}_j]$. The first message is called responsibility $r(\mathbf{o}_i, \mathbf{o}_j)$, which measures the accumulated evidence that $\mathbf{o}_j$ should be the exemplar for $\mathbf{o}_i$. Formally, the responsibility is given as in Equation 6.6.

$$r(\mathbf{o}_i, \mathbf{o}_j) \leftarrow md(\mathbf{o}_i, \mathbf{o}_j) - \max_{\forall \mathbf{o}'_j \neq \mathbf{o}_j} \left\{ \alpha(\mathbf{o}_i, \mathbf{o}'_j) + md(\mathbf{o}_i, \mathbf{o}'_j) \right\} \tag{6.6}$$

The availability α of an element $\mathbf{o}_j$ to be the exemplar of $\mathbf{o}_i$ is given as in Equation 6.7.

$$\alpha(\mathbf{o}_i, \mathbf{o}_j) \leftarrow \min \left\{ 0, r(\mathbf{o}_j, \mathbf{o}_j) + \sum_{\mathbf{o}'_i \text{ s.t. } \mathbf{o}'_i \notin \{\mathbf{o}_i, \mathbf{o}_j\}} \max \left\{ 0, r(\mathbf{o}'_i, \mathbf{o}_j) \right\} \right\} \tag{6.7}$$

Responsibility r and availability α are initially zero, and all $\mathbf{o}_i$, $\mathbf{o}_j$ equally represent a potential exemplar FD. At any time of affinity propagation, measures r and α can be combined to identify exemplars functional dependencies. Responsibility iteration lets all elements $\mathbf{o}_i$ compete for ownership of another $\mathbf{o}_j$, and availability iterations choose evidence for every other element $\mathbf{o}_i$ as to whether each candidate exemplar would make a satisfying exemplar FD.

FDSEL iterate the input Σ to find the corresponding right-hand side of the affinity propagation output. This set of exemplars functional dependencies is also extended with inference rules.

6.2.5 Semantic Query Optimization

We present a scenario in which the selected functional dependencies can improve the overall query performance. We use the approach presented in [150] and [151], nevertheless, our tool can be extended to work with others dependency-aware optimization schemes like [152] and [153]. Each optimization rewrites the incoming queries into syntactically different, yet semantically equivalent queries. The rewritten queries are semantically equivalent if and only if their results are the same as the original query, regardless of the state of the database [151]. The rewritten queries are expected to produce a more efficient execution plan.

Rewritings could be blocked for particular queries according to the trade-off between optimization time and the quality of the execution strategies. As noted by [154], semantic

optimization increases the search space of possible plans and, as a result, relies on efficient searching techniques to keep optimization costs within reasonable bounds. FDSEL is decoupled from any database management systems query optimizer, and its first and foremost goal is to select suitable functional dependencies for optimization. Thus, the incoming queries are only rewritten when they fall into two distinct classes. FDSEL uses the exemplars functional dependencies to carry out two classes of semantic query optimization commonly discussed in the literature [150, 151, 152, 153]: join elimination and order optimization.

The join elimination technique iterates over the set of functional dependencies to find residual clauses in the query. In this case, residuals clauses are joins for which the result is known a priori (empty or redundant joins) and, therefore, could be removed from the query. Consider a relation $\mathsf{R} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$, and a functional dependency $f : \mathsf{A} \rightarrow \mathsf{B}$ holding in an instance r of R. The relation R can be decomposed as $\mathsf{R}' = \pi_{(\mathsf{A},\mathsf{B})}(\mathsf{R})$, and $\mathsf{R}'' = \pi_{(\mathsf{A},\mathsf{C})}(\mathsf{R})$. This lossless-join decomposition is used to target queries where no attributes are selected or projected from the R'' relation. The join elimination optimization is already implemented in some commercial database management systems [125]. However, these implementations require the users to declare the set of constraints explicitly. Thus, automating this task may be beneficial in environments where users access views defined over a large number of joins (e.g., a star schema in a data warehouse). Further details and more complex join elimination optimizations can be found in [151] and [125].

The goal of order optimization is to find optimal sorting orders, that is, the best sequence of the attributes in the order specification. Sorting orders usually emerge when tables are joined; or when tuples are ordered, grouped, or distinguished. The algorithm presented in [150] takes as input a set of functional dependencies, a set of predicates, and sorting orders specifications to return an optimized sorting order specification. Consider a functional dependency $f : X \rightarrow Y$ holding on relation instance r, and a query $\mathsf{q} = \tau_{X,Y}\left(\pi_{(X,Y)}(\mathsf{R})\right)$. The query q can be rewritten as $\mathsf{q}' = \tau_{X}\left(\pi_{(X,Y)}(\mathsf{R})\right)$ because there is only one value of Y for each X. More examples and details on order optimization can be found in [150] and [59].

6.3 EXPERIMENTAL STUDY

In this section, we present an experimental study to evaluate the effectiveness of FDSEL.

6.3.1 Scenario

The use of functional dependencies for semantic query optimization can provide compelling gains in environments where relations are vertically partitioned (e.g., column-stores in data warehouses). In practice, there are many reasons why partitioning may be required. For example, database administrators might fragment a relation into a set of smaller relations to reduce maintenance costs, or to cope with distributed designs where applications use some fragments more frequently than others (e.g., invisible joins in column-stores [155]). Another example is normalization (e.g., to Boyce-Codd Normal Form), which uses functional dependencies to

eliminate redundancies and anomalies introduced as the dataset grow [102]. Regardless of the reasons for table partitioning, a typical mechanism to reconstitute information from partitions is *views*. Views can be defined using arbitrarily complex queries that blindly join partitions in order to present the user with a representation of the original table, with potential restrictions. The users' access may be limited to the defined views (maybe through a query manager interface); therefore, redundant joins or residual sorting order operations are likely to occur. We use the scenario based on views to present our experimental evaluation.

6.3.2 Datasets and implementation details

Datasets. We use both synthetic and real-world datasets, which come from different domains. Table 6.2 lists these datasets with their number of attributes, number of records, number of discovered functional dependencies, and number of exemplars selected according to the three selection strategies of FDSEL. The datasets *Abalone* and *Adults* have been used for functional dependency discovery evaluation in [27]. The *Adults* dataset is based on census data for US citizen salaries. The *Abalone* dataset consists of clinical data about patients and diseases. In addition, we use a 2-week snapshot of data extracted by *SIMMC*, a brazilian project from the Ministry of Communications [156] [1]. SIMMC dataset has about 2M records with a total size of nearly 300MB. Finally, we use the *lineitem* relation of the business-oriented synthetic TPC-H dataset, set for a 1GB scale.

Implementation details. We executed our experiments on a single machine with a 2.60 GHz Quad Core i7-3720QM processor, 8GB of RAM, 500GB 7200rpm SATA II disk, and Java 1.8. The machine runs Ubuntu 16.04. Our prototype is a Java client that connects to a PostgreSQL server via JDBC.

FDSEL discovers the set of functional dependencies holding on each dataset and stores the results in a buffer. After the discovery, we use a tool called *Normalize* to decompose the original dataset into a set of tables that is BCNF-conformed [102] . We supervise the results of *Normalize* to avoid semantically incorrect partitions. During our experiments, this partitioning step generated between three and six tables for each dataset. These tables are joined at random to build the set of views in which queries run.

We execute select-project-join queries and select-project-join with group by queries over the views, which are chosen at random. To choose the range of filter predicates, we equally divide the domain of each attribute according to the number of queries N to be executed. If the number of distinct values in the attribute domain is less than N, we assume the sequence of the closest pairs of values in the domain. For these cases, overlapping query predicates is required. Predicate ranges are chosen using a Zipfian distribution on the number of queries N [157]. We also follow a Zipfian distribution to choose attributes for selections, projections, and grouping. We vary the number of attributes in each operation according to the number of attributes in the

[1] http://simmc.c3sl.ufpr.br/

Table 6.2: Description of the datasets, number of functional dependencies (FDs), and number of exemplars functional dependencies with FDSEL.

Dataset	#Columns	#Records	#FDs	#FDs with FDSEL - Criterion 1	#FDs with FDSEL - Criterion 2	#FDs with FDSEL - affinity propagation
Abalone	9	4,177	137	6	6	10
Adults	14	48,842	78	9	3	5
SIMMC	12	2m	32	5	3	8
Lineitem	16	6m	4k	8	101	23

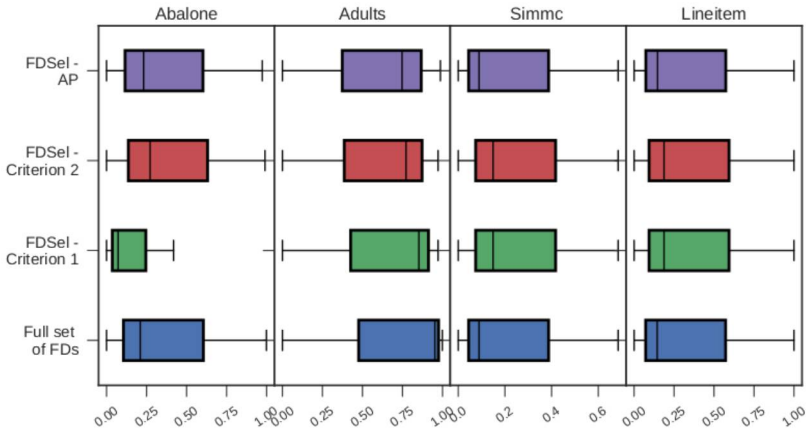
view. Finally, we use the same distribution configuration to generate a thousand queries for the training workload and a hundred queries for performance evaluation with semantic query optimization. We use the above procedures to run FDSEL set for either join elimination or order optimization, and we report their performance results separately.

The training workload and functional dependency buffer comprise the input of FDSEL. The core component selects the exemplars of functional dependencies and prepares the query mediator for optimizations. The query mediator is conditioned to the semantics of each query. If the set of operations and attributes required to evaluate the query fall into rules conforming to join elimination or order optimization (based on the set of exemplars functional dependencies), it rewrites the query; otherwise, it bypasses the rewriting process.

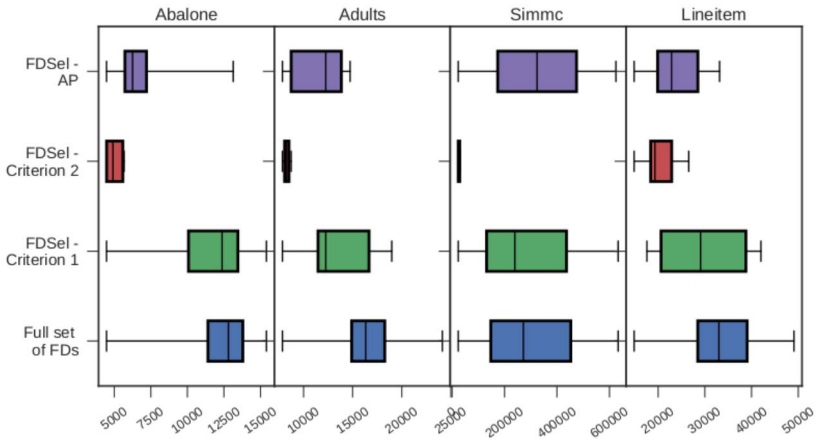
6.3.3 Effectiveness

Selecting functional dependencies is subjective to a combination of factors (e.g., application, schema-level structures, and instance-level information). Also, the number of discovered functional dependencies are usually too large for manual inspection. For the following results, we evaluate the quality of the exemplar functional dependencies based on their quality according to the measures described in Section 6.2, and their suitability for semantic query optimization.

We estimated quality measures for the sets of exemplars functional dependencies returned by Algorithm 8, pruned with Criterion 1 (increase in distrust); Algorithm 8, pruned with Criterion 2 (increase in Mahalanobis distance); and affinity propagation clustering algorithm. We refer to these results as FDSEL - Criterion 1, FDSEL - Criterion 2, and FDSEL - affinity propagation, respectively. In addition, we estimated the quality measures for the initial set of discovered functional dependencies to form a baseline. For each Denial constraint discovery algorithms, we estimated its *distrust* level and its Mahalanobis distance from the query workload. Figure 6.2 shows the distributional characteristics of the quality measures in a box-and-whisker plot. Each box divides the measures estimated for a set of functional dependencies into quartiles to illustrate their degree of concentration and range. The bottom boxes and whiskers (we refer to them as bases) show the concentration and range of the measures for the set of initial functional dependencies and serve as the reference point for assessing which way the results from FDSEL sway.



(a) Levels of distrust



(b) *Mahalanobis* distances between functional dependencies and workload

Figure 6.2: Quality of exemplar functional dependencies. The bottom boxes represent the distributional trends for the initial set of functional dependencies. The remaining boxes represent the distributional trends of the exemplar functional dependencies returned by FDSEL.

In general, the results from FDSEL procedures were fairly close to that of the bases for *distrust* (Figure 6.2(a)). FDSEL - Criterion 1 presented more pronounced gains only for Abalone, where distribution measures are thicker and closer to the lowest values. For other datasets and strategies, FDSEL causes slight alterations at the center quartiles. In a more in-depth analysis, we have found that many functional dependencies exhibit similar levels of *distrust*. These functional dependencies form groups that are easily distinguished by their attributes (e.g., functional dependencies with many attributes in common at their left-hand side). Because we rank the set of functional dependencies regarding structural frequencies (left-hand side and right-hand side) weighted over the workload, similar functional dependencies are likely to be sorted into close spots at the sequence. However, the lack of a single attribute at their structure may cause *distrust* to change dramatically.

As can be seen in Figure 6.2(b), the distributions for Mahalanobis distance reveal much more pronounced variations. That is because the initial set of functional dependencies present different levels of correlation to the query workload, and because FDSEL uses different strategies to select exemplars functional dependencies. For Criterion 1, FDSEL may start discarding relevant functional dependencies sooner than other procedures (e.g, contrast between *distrust* and Mahalanobis distance in Abalone).

FDSEL - Criterion 2 produced the best Mahalanobis distance distributions. It softens the *distrust* barrier from FDSEL - Criterion 1 and focus on the Mahalanobis distance of each functional dependency. FDSEL - Criterion 2 was able to produce distance measures that concentrate towards lower values (all quartile groups spread themselves to the first half of the distribution). Notably, it was the most effective procedure when the number of original functional dependencies was relatively small. For SIMMC, all exemplars exhibit distance measures that are close to the lower tail of the distribution. Nevertheless, if the number of functional dependencies is high, the distances for the set of original functional dependencies approximate normal distributions (e.g., Lineitem). FDSEL - Criterion 2 selects exemplars that are more likely to fall closer to minimum values for Mahalanobis distance. As a result, it may disregard groups of functional dependencies with higher distances but also higher semantics (e.g., a high number of correlated attributes at the left-hand side). That might occur if functional dependencies have a higher number of attributes. Because of their weighted equivalence, functional dependencies with more attributes may increase the likelihood of larger Mahalanobis distances.

Criterion 1 and 2 may become over-judicious for some base distributions and discard relevant functional dependencies. The selection task should achieve parsimony between the number of exemplars and the semantics they expose because such characteristic is compelling in the optimization phase. The distributions for FDSEL - affinity propagation suggests that the exemplars have proper levels of agreement with the workload, leaning reach, and distributions toward the first half of the base (except by SIMMC dataset). Interestingly, the exemplars for the SIMMC produced more uniform distributions if compared to the base. Though the original set

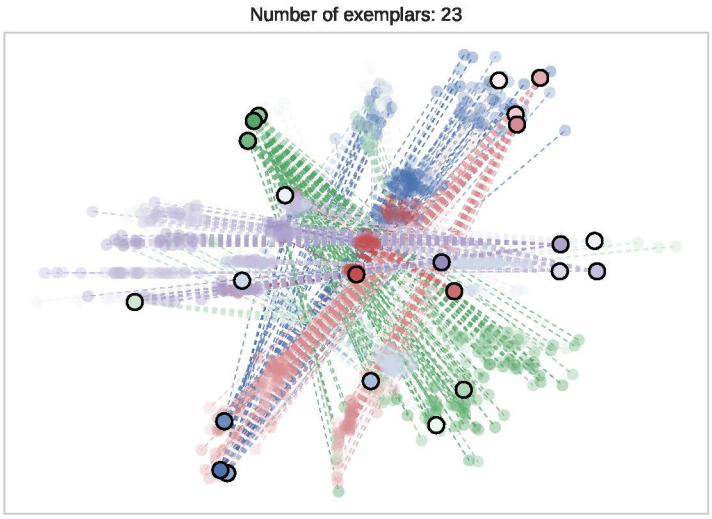


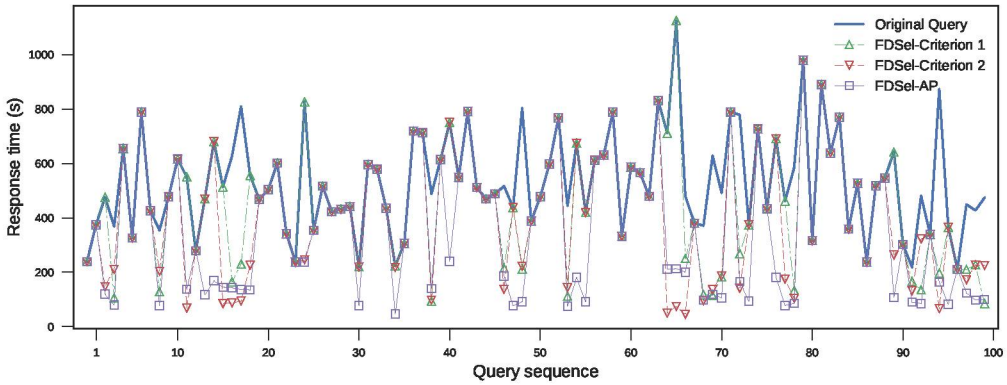
Figure 6.3: Behavior of FDSEL - Affinity Propagation over Lineitem dataset. Dimensions were reduced with Principal Component Analysis for better visualization.

of functional dependencies had just a few distance measures concentrated at the fourth quartile, FDSEL - affinity propagation was able to select exemplars from it. That was only possible because of the intrinsic characteristic of the affinity propagation algorithm in combination with the *Mahalanobis* distance. As described in Section 6.2, the affinity propagation algorithm simultaneously considers any functional dependency in the original set as a possible exemplar. Because affinity propagation refines this large set by exchanging similarity messages between its elements (functional dependencies), it was crucial to choose a distance measure that could capture the semantic aspects of an functional dependency along with the workload closeness. With Mahalanobis distance, the similarities between pairs of functional dependencies in the space are defined by the weighted attributes.

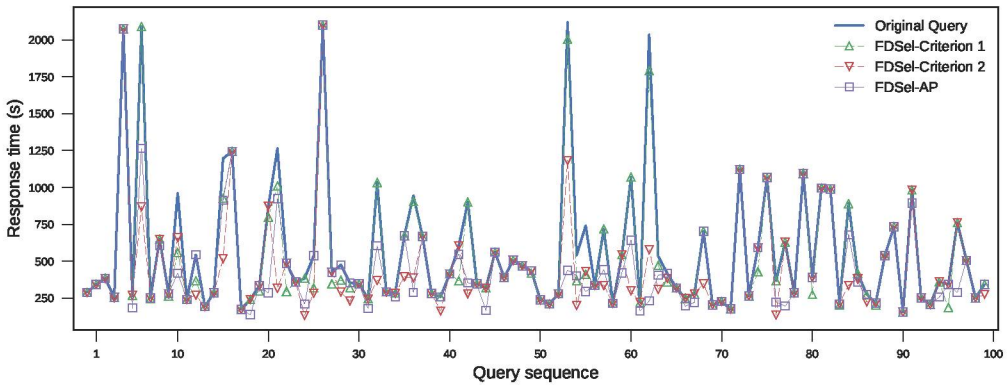
Because Mahalanobis distance accounts for unequal variances and correlations between the weighted attributes, it estimates the distances by assigning different influence factors to the attributes in each functional dependency. Differently from Criterion 2, the selection with affinity propagation not only considers distance values but also considers how many sets of attributes are correlated.

Figure 6.3 represents the overall behavior of affinity propagation applied over the functional dependencies of *lineitem* relation. Notice that exemplars can be responsible for representing distant data points. That is why affinity propagation was able to select exemplars from spread locations in the distribution but, at the same time, shortening the range of distances with the workload.

Table 6.2 reports the number of all discovered functional dependencies and the number of exemplars selected by FDSEL. As we shall see in the next experiment, a high number of



(a) Join Elimination



(b) Order Optimization

Figure 6.4: Example of improvements in query execution time with FDSEL over lineitem.

exemplars does not necessarily mean better optimizations and, therefore, does not guarantee higher gains in query performance.

6.3.4 Performance improvement with semantic query optimization

In this experiment, we investigate the performance improvements of using FDSEL for semantic query optimization. Figures 6.4(a) and 6.4(b) illustrate the implication of join elimination and order optimization optimizations for queries running over *lineitem*. The execution time remains unchanged for some queries because FDSEL could not find any rewrite strategy for them. However, improvements of more than an order of magnitude can be viewed for many queries.

As expected, join elimination showed the most significant reductions in execution time for best-cases. For example, a particular query over *lineitem* reached a 12-fold improvement with FDSEL - affinity propagation. FDSEL - affinity propagation presented bigger improvements for larger datasets (SIMMC and Lineitem) because, for some predicates, the queries produced intermediary results that do not fit in main memory. Although order optimization produced moderate improvements, the number of queries that benefited from rewriting was more consistent. For example, there were many queries in SIMMC that reached 4 to 6-fold improvements. Tables

6.3 and 6.4 details the average improvements with join elimination and order optimization, respectively. On average, approximately one-third of the workload (for all datasets) was able to take advantage of some rewriting rules. FDSEL was able to reduce the average execution time by nearly half in many cases.

Some of the best improvements occurred when the number of exemplars available was among the smallest (SIMMC and FDSEL - Criterion 2). This fact confirms our hypothesis that focusing on the information implied by the context usage (e.g., query workload) is more effective than necessarily considering a large number of functional dependencies. For example, FDSEL - affinity propagation over the queries in *lineitem* presented the best performance even though it relied on less than a quarter of the number of exemplars of FDSEL - Criterion 2.

Table 6.3: Performance improvements with FDSEL in join elimination.

Dataset	Normal Execution (Avg)	FDSEL - Criterion 1 (Avg)	FDSEL - Criterion 2 (Avg)	FDSEL - affinity propagation (Avg)
Abalone	22ms	18ms	17ms	15ms
Adults	209ms	152ms	136ms	123ms
SIMMC	22.90s	15.79s	12.03s	13.44s
Lineitem	531.33s	383.51s	344.75s	297.10s

6.4 SUMMARY

Dependencies among data permeate databases, and, whenever possible, should be exploited in data management tasks. Although several commercial solutions present facilities to unite not enforced constraints (such as functional dependencies) into planning phases, we cannot expect them to be exploited in query plans without human supervision. In this chapter, we present FDSEL, an automatic tool for selecting functional dependencies in relational databases. FDSEL is based on the idea of matching functional dependencies with the current workload to boost query optimization. First, we model attribute occurrence matrices (AOMs) with the functional dependencies and the workload information. We provide operations over the AOMs to estimate weights over each matching. Then, we present strategies to investigate this matching: (1) ranking functional dependencies that match most of the projections/selections in the query

Table 6.4: Performance improvements with FDSEL in order optimization.

Dataset	Normal Execution (Avg)	FDSEL - Criterion 1 (Avg)	FDSEL - Criterion 2 (Avg)	FDSEL - affinity propagation (Avg)
Abalone	40ms	38ms	24ms	24ms
Adults	367ms	320ms	220ms	237ms
SIMMC	62s	54s	35s	29s
Lineitem	571s	487s	387s	360s

stream (i.e., workload); and (2) clustering functional dependencies on their *lhs* structure, with only the most representative matching elements set as exemplars. Next, we compute the distance between binary relationships of functional dependencies and workload to focus on well-ranked functional dependencies (by the ranking strategy) or similar ones (by the clustering strategy). Finally, we indicate the focused exemplars in hand to help with semantic query optimizations.

The results from both ranking and clustering strategies showed that FDSEL can find sets of functional dependencies with *distrust* distributions reasonably similar to those produced by the exhaustive functional dependencies discovery approaches. The results also demonstrated the effectiveness of FDSEL at discovering functional dependencies on different datasets (one of them running in production) for query optimization, frequently reducing query response time is up to 1 order of magnitude in join elimination.

Chapter 7

Conclusions

This thesis presents a novel data profiling algorithm for denial constraints, introduces diverse approaches that help in applying denial constraints for the improvement of data quality, and describes a system for the application of functional dependencies in query optimization. The list of publications we contributed during the development of this thesis is available in Appendix A.

The challenges in discovering approximate denial constraints drove us to design DCFINDER algorithm. We can take several algorithmic insights from it. The combination of position list indexes, logical operations, and predicate selectivity results in a time-efficient building of evidence sets. This building step is critical in denial constraint discovery since the algorithms for the task explore the search space and validate candidates using evidence sets. Also, DCFINDER was designed to maintain complete information on evidence multiplicity, which is required in discovering approximate denial constraints. In our experimental evaluation, the design decisions in DCFINDER showed to improve the runtime of denial constraint discovery considerably.

This thesis also shows that the evidence distribution taken from a given consistent dataset differs from the evidence distribution taken from an equivalent dataset containing some inconsistencies. In that context, this thesis presents a method based on evidence multiplicity that extends DCFINDER to discover reliable approximate denial constraints from inconsistent data. The approach is promising because the access to 100% consistent data is often infeasible. Our evaluation showed that our method discovers approximate denial constraints that identify many inconsistencies in the input dataset.

We saw that current commercial database management systems might take too long in detecting violations of denial constraints commonly seen in production. This thesis introduces VIOFINDER to handle this detection problem efficiently. We learned that combining pipelines of tuple partitions with refinement implementation based on predicate type bring a fast execution of violation detection, at a relatively low memory footprint. Being the fastest option in our experimental evaluation, VIOFINDER can be a compelling component for any data cleaning pipeline or tool based on denial constraints.

Finally, this thesis describes a system that uses functional dependencies discovered from datasets to improve query optimization. FDSEL explores query workload information to narrow a large number of functional dependencies to those that can benefit the most from query rewritings based on join elimination or order optimization. In our experimental analysis, we found that FDSEL can frequently apply query rewritings to reduce overall query response time.

7.1 FINAL THOUGHTS AND FUTURE WORKS

We start this section with a brief discussion on the scalability of data profiling algorithms for the discovery of dependencies. The evaluation of DCFINDER and other related algorithms shows that discovering approximate denial constraints may take hours for relatively small datasets (with around one million of records and two dozen columns). Such long runtime appears even if we consider the discovery of exact denial constraints, which enables a series of optimizations in its algorithms. A long runtime also appears in the discovery of dependencies that are simpler than denial constraints; see, for example, the experimental evaluation on functional dependency discovery in [29]. These performance results are somehow expected, as they only reflect the computational complexity of the dependency discovery problems.

One approach that can reduce runtime for dependency discovery is sampling [158, 159, 160]. For example, in [160], the authors adapt DCFINDER to work with data samples and show that it is possible to reduce the discovery of denial constraints runtime at small completeness sacrifices. When considering sampling, the problem becomes that of designing methods that can guarantee some completeness bounds. Even though this line of research is orthogonal to the one presented in this thesis, we believe it is a promising approach that can help with several scalability issues in profiling dependencies. Unfortunately, even the use of sampling in dependency discovery might be undermined because the output can be quite large due to the exponential nature of the discovery problem. This fact, however, does not indicate that we have reach a dead end.

Production applications would hardly require a large number of dependencies, such as the number of dependencies in dependency discovery output, in their operation. Besides, we saw that the number of denial constraints with high coverage and succinctness is relatively small. Also, we noticed that the number of functional dependencies that benefit query optimization the most is relatively small as well. Based on these facts, focusing the dependency discovery on the subset of results that are eventually applied in applications might be explored to enable the profiling of more massive datasets.

An exciting line for future research regards dynamic data. Datasets receive data updates continually, and as a result, their data profiles change regularly. Solutions that can discover dependencies or detect dependency violations while datasets are changing are quite helpful because it would avoid the re-execution of the long-running processes in the entire data. There has been recent research around these lines [161, 30, 162]. As static solutions have inspired

these methods, we believe the static approaches we describe in this thesis can be a starting point for discovering denial constraints and detecting violations on dynamic data.

REFERENCES

[1] Wenfei Fan. Data quality: From theory to practice. *SIGMOD Rec.*, 44(3):7–18, December 2015.

[2] Michael M. Hammer and Dennis J. McLeod. Semantic integrity in a relational data base system. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, page 25–47, New York, NY, USA, 1975. Association for Computing Machinery.

[3] Michael Stonebraker. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data*, SIGMOD '75, page 65–78, New York, NY, USA, 1975. Association for Computing Machinery.

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

[5] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

[6] W. W. Eckerson. Data quality and the bottom line: achieving business success through a commitment to high quality data. Technical report, Data Warehousing Institute, 2002.

[7] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE TVCG*, 18(12), Dec 2012.

[8] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2 edition, 2008.

[9] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

[10] Umeshwar Dayal, Jennifer Widom, and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, USA, 1994.

[11] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, March 1999.

[12] Roberta Cochrane, Hamid Pirahesh, and Nelson Mendonça Mattos. Integrating triggers and declarative constraints in sql database sytems. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, page 567–578, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[13] Eric Simon and Angelika Kotz Dittrich. Promises and realities of active database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, page 642–653, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[14] Stefano Ceri, Roberta Cochrane, and Jennifer Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 254–262, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[15] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3 edition, 2002.

[16] V. Wiktor Marek and Miroslaw Truszczynski. Revision programming, database updates and integrity constraints. In Georg Gottlob and Moshe Y. Vardi, editors, *Database Theory - ICDT'95, 5th International Conference, Prague, Czech Republic, January 11-13, 1995, Proceedings*, volume 893 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 1995.

[17] Wojciech Czerwinski, Claire David, Filip Murlak, and Pawel Parys. Reasoning about integrity constraints for tree-structured data. In Wim Martens and Thomas Zeume, editors, *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, volume 48 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[18] Nishita Balamuralikrishna, Yingnan Jiang, Henning Koehler, Uwe Leck, Sebastian Link, and Henri Prade. Possibilistic keys. *Fuzzy Sets and Systems*, 376:1 – 36, 2019. Theme: Computer Science.

[19] Batya Kenig and Dan Suciu. Integrity constraints revisited: From exact to approximate implication. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*, volume 155 of *LIPIcs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[20] Ronald Fagin and Moshe Y. Vardi. The theory of data dependencies — an overview. In Jan Paredaens, editor, *Automata, Languages and Programming*, pages 1–22, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[21] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data - a review. *IEEE TKDE*, 24(2):251–264, February 2012.

[22] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: A survey. *The VLDB Journal*, 24(4):557–581, August 2015.

[23] L. Caruccio, V. Deufemia, and G. Polese. Relaxed functional dependencies - a survey of approaches. *IEEE TKDE*, 28(1):147–165, Jan 2016.

[24] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2):6:1–6:48, June 2008.

[25] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. Discovering conditional functional dependencies. *IEEE TKDE*, 23(5):683–698, May 2011.

[26] Yka Huhtala, Juha Karkkainen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100, 1999.

[27] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *PVLDB.*, 8(10):1082–1093, June 2015.

[28] Z. Wei and S. Link. Discovery and ranking of functional dependencies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1526–1537, 2019.

[29] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 821–833, New York, NY, USA, 2016. ACM.

[30] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. Dynfd: Functional dependency discovery in dynamic datasets. In Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi, editors, *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 253–264. OpenProceedings.org, 2019.

[31] Laure Berti-Équille, Hazar Harmouch, Felix Naumann, Noël Novelli, and Saravanan Thirumuruganathan. Discovery of genuine functional dependencies from relational data with missing values. *Proc. VLDB Endow.*, 11(8):880–892, April 2018.

[32] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509, August 2013.

[33] Matteo Magnani and Danilo Montesi. A survey on uncertainty management in data integration. *J. Data and Information Quality*, 2(1):5:1–5:33, July 2010.

[34] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB Endow.*, 10(11):1190–1201, August 2017.

[35] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. Nadeef: A commodity data cleaning system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 541–552, New York, NY, USA, 2013. ACM.

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

[37] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.

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

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

[40] Leo L. Pipino, Yang W. Lee, and Richard Y. Wang. Data quality assessment. *Commun. ACM*, 45(4):211–218, April 2002.

[41] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. Discovery of approximate (and exact) denial constraints. *Proc. VLDB Endow.*, 13(3):266–278, November 2019.

[42] Eduardo H. M. Pena and Eduardo Cunha de Almeida. Short paper: Descoberta automática de restrições de negação confiáveis. In *XXXIV Simpósio Brasileiro de Banco de Dados, SBBD 2019, Fortaleza, CE, Brazil, October 7-10, 2019*, pages 187–192. SBC, 2019.

[43] Eduardo H. M. Pena, Edson R. Lucas Filho, Eduardo C. de Almeida, and Felix Naumann. Efficient detection of data dependency violations. to appear in Proceedings of the 29th ACM International Conference on Information and Knowledge Management, 2020.

[44] Eduardo H. M. Pena, Erik Falk, Jorge Augusto Meira, and Eduardo Cunha de Almeida. Mind your dependencies for semantic query optimization. *JIDM*, 9(1):3–19, 2018.

[45] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[46] E. F. Codd. Further normalization of the data base relational model. *Research Report / RJ / IBM / San Jose, California*, RJ909, 1971.

[47] E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972.

[48] Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, page 159–170, New York, NY, USA, 2008. Association for Computing Machinery.

[49] Thomas J. Watson IBM Research Center and M.Y. Vardi. *Fundamentals of Dependency Theory*. Research report. IBM Research Division, 1985.

[50] Paolo Atzeni and Valeria De Antonellis. *Relational database theory*. Benjamin-Cummings Publishing Co., Inc., 1993.

[51] E.O. [de Brock]. A general treatment of dynamic integrity constraints. *Data and Knowledge Engineering*, 32(3):223 – 246, 2000.

[52] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '82, page 171–176, New York, NY, USA, 1982. Association for Computing Machinery.

[53] Guido Geerts. Semantic modelling of an accounting universe of discourse: The usefulness of inter-relation constraints. In Toby J. Teorey, editor, *Proceedings of the 10th International Conference on Entity-Relationship Approach (ER'91), 23-25 October, 1991, San Mateo, California, USA*, pages 263–283. ER Institute, 1991.

[54] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. pages 458–469, 2013.

[55] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. Efficient denial constraint discovery with hydra. *Proc. VLDB Endow.*, 11(3):311–323, November 2017.

[56] Marianne Baudineta, Jan Chomicki, and Pierre Wolper. Constraint-generating dependencies. *Journal of Computer and System Sciences*, 59(1):94 – 115, 1999.

[57] Wenfei Fan and Floris Geerts. *Foundations of Data Quality Management*. Morgan and Claypool Publishers, 2012.

[58] Wilfred Ng. Ordered functional dependencies in relational databases. *Information Systems*, 24(7):535 – 554, 1999.

[59] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Expressiveness and complexity of order dependencies. *Proc. VLDB Endow.*, 6(14):1858–1869, September 2013.

[60] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. Fundamentals of order dependencies. *Proc. VLDB Endow.*, 5(11):1220–1231, July 2012.

[61] Seymour Ginsburg and Richard Hull. Order dependency in the relational model. *Theoretical Computer Science*, 26(1):149 – 195, 1983.

[62] Jyrki Kivinen and Heikki Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129 – 149, 1995.

[63] Batya Kenig and Dan Suciu. Integrity Constraints Revisited: From Exact to Approximate Implication. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory (ICDT 2020)*, volume 155 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:20, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[64] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. Metric functional dependencies. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1275–1278, Washington, DC, USA, 2009. IEEE Computer Society.

[65] Shaoxu Song and Lei Chen. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.*, 36(3):16:1–16:41, August 2011.

[66] Sridevi Baskaran, Alexander Keller, Fei Chiang, Lukasz Golab, and Jaroslaw Szlichta. Efficient discovery of ontology functional dependencies. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 1847–1856, New York, NY, USA, 2017. ACM.

[67] Xiaoyuan Wang, Xingzhi Sun, Feng Cao, Li Ma, Nick Kanellos, Kang Zhang, Yue Pan, and Yong Yu. Smdm: Enhancing enterprise-wide master data management using semantic web technologies. *Proc. VLDB Endow.*, 2(2):1594–1597, August 2009.

[68] Karin Murthy, Prasad M. Deshpande, Atreyee Dey, Ramanujam Halasipuram, Mukesh Mohania, P. Deepak, Jennifer Reed, and Scott Schumacher. Exploiting evidence from

unstructured data to enhance master data management. *Proc. VLDB Endow.*, 5(12):1862–1873, August 2012.

[69] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. Towards certain fixes with editing rules and master data. *The VLDB Journal*, 21(2):213–238, April 2012.

[70] Jiannan Wang and Nan Tang. Dependable data repairing with fixing rules. *J. Data and Information Quality*, 8(3-4):16:1–16:34, June 2017.

[71] Sergio Greco, Cristian Molinaro, and Francesca Spezzano. *Incomplete Data and Data Dependencies in Relational Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[72] Ziheng Wei and Sebastian Link. Embedded functional dependencies and data-completeness tailored database design. *Proc. VLDB Endow.*, 12(11):1458–1470, July 2019.

[73] Z. Wei, U. Leck, and S. Link. Entity integrity, referential integrity, and query optimization with embedded uniqueness constraints. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1694–1697, 2019.

[74] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. *Data Profiling*. Morgan and Claypool Publishers, 2018.

[75] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combinations. 7(4):301–312, 2013.

[76] Noel Novelli and Rosine Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, pages 189–203, London, UK, UK, 2001. Springer-Verlag.

[77] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '00, pages 350–364, London, UK, UK, 2000. Springer-Verlag.

[78] Catharine Wyss, Chris Giannella, and Edward L. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In *Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, DaWaK '01, pages 101–110, London, UK, UK, 2001. Springer-Verlag.

[79] C. Giannella and C.M. Wyss. Finding minimal keys in a relation instance, 1999.

[80] Ziawasch Abedjan and Felix Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, page 1565–1570, New York, NY, USA, 2011. Association for Computing Machinery.

[81] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. Gordian: Efficient and scalable discovery of composite keys. pages 691–702. ACM, 2006.

[82] Thorsten Papenbrock and Felix Naumann. A hybrid approach for efficient unique column combination discovery. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[83] Philipp Langer and Felix Naumann. Efficient order dependency detection. *The VLDB Journal*, 25(2):223–241, April 2016.

[84] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *Proc. VLDB Endow.*, 10(7):721–732, March 2017.

[85] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. Discovering order dependencies through order compatibility. pages 409–420, 2019.

[86] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Erratum for discovering order dependencies through order compatibility (EDBT 2019). In Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang, editors, *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 659–663. OpenProceedings.org, 2020.

[87] Y. Jin, L. Zhu, and Z. Tan. Efficient bidirectional order dependency discovery. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 61–72, 2020.

[88] Mohammed J. Zaki. Mining non-redundant association rules. *Data Min. Knowl. Discov.*, 9(3):223–248, November 2004.

[89] Joeri Rammelaere and Floris Geerts. Revisiting conditional functional dependency discovery: Splitting the "C" from the "FD". In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 552–568, 2019.

[90] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. Functional and approximate dependency mining: Database and fca points of view. *J. Exp. Theor. Artif. Intell.*, 14:93–114, 04 2002.

[91] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. 11(7):759–772, 2018.

[92] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of bidirectional order dependencies via set-based axioms. 27(4):573–591, 2018.

[93] S. Song, L. Chen, and H. Cheng. Efficient determination of distance thresholds for differential dependencies. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2179–2192, 2014.

[94] Selasi Kwashie, Jixue Liu, Jiuyong Li, and Feiyue Ye. Efficient discovery of differential dependencies through association rules mining. In Mohamed A. Sharaf, Muhammad Aamir Cheema, and Jianzhong Qi, editors, *Databases Theory and Applications*, pages 3–15, Cham, 2015. Springer International Publishing.

[95] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Advances in knowledge discovery and data mining. chapter Fast Discovery of Association Rules, pages 307–328. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[96] Thierno Diallo, Jean-Marc Petit, and Sylvie Servigne. Discovering Editing Rules for Data Cleaning. In *10th International Workshop on Quality in Databases In conjunction with VLDB (Very Large Databases) 2012*, pages 1–8, Istanbul, Turkey, August 2012.

[97] Ziheng Wei, Sven Hartmann, and Sebastian Link. Discovery algorithms for embedded functional dependencies. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 833–843, New York, NY, USA, 2020. Association for Computing Machinery.

[98] Ziheng Wei, Uwe Leck, and Sebastian Link. Discovery and ranking of embedded uniqueness constraints. *Proc. VLDB Endow.*, 12(13):2339–2352, September 2019.

[99] Gregory Piatetsky-Shapiro and Christopher J. Matheus. Measuring data dependencies in large databases. In *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases*, AAAIWS'93, page 162–173. AAAI Press, 1993.

[100] Daniel Sánchez, José-María Serrano, Ignacio J. Blanco, María J. Martín-Bautista, and María Amparo Vila Miranda. Using association rules to mine for strong approximate dependencies. *Data Min. Knowl. Discov.*, 16(3):313–348, 2008.

[101] Periklis Andritsos, Renée J. Miller, and Panayiotis Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *Proceedings of the 2004*

ACM SIGMOD International Conference on Management of Data, SIGMOD '04, page 731–742, New York, NY, USA, 2004. Association for Computing Machinery.

[102] Thorsten Papenbrock and Felix Naumann. Data-driven schema normalization. pages 342–353, 2017.

[103] B. Saha and D. Srivastava. Data quality: The other face of big data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1294–1297, March 2014.

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

[105] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. That's all folks!: Llunatic goes open source. pages 1565–1568, 2014.

[106] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. Bigdansing: A system for big data cleansing. In *SIGMOD*, page 1215–1230, 2015.

[107] Leopoldo Bertossi. Consistent query answering in databases. *SIGMOD Rec.*, 35(2):68–76, June 2006.

[108] Balder ten Cate, Gaëlle Fontaine, and Phokion G. Kolaitis. On the data complexity of consistent query answering. *Theory of Computing Systems*, 57(4):843–891, Nov 2015.

[109] Marco Calautti, Leonid Libkin, and Andreas Pieris. An operational approach to consistent query answering. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS '18, pages 239–251, New York, NY, USA, 2018. ACM.

[110] Sebastian Arming, Reinhard Pichler, and Emanuel Sallinger. Complexity of Repair Checking and Consistent Query Answering. In Wim Martens and Thomas Zeume, editors, *19th International Conference on Database Theory (ICDT 2016)*, volume 48 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:18, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[111] Gaëlle Fontaine. Why is it hard to obtain a dichotomy for consistent query answering? *ACM Trans. Comput. Logic*, 16(1):7:1–7:24, March 2015.

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

[113] 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 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 143–154, New York, NY, USA, 2005. ACM.

[114] Gao Cong, 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 '07, pages 315–326. VLDB Endowment, 2007.

[115] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using deepdive. *Proc. VLDB Endow.*, 8(11):1310–1321, July 2015.

[116] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008.

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

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

[119] George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. On the relative trust between inconsistent data and inaccurate constraints. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 541–552, 2013.

[120] Mirjana Mazuran, Elisa Quintarelli, Letizia Tanca, and Stefania Ugolini. Semi-automatic support for evolving functional dependencies. In *EDBT 2016.*, pages 293–304, 2016.

[121] Michael Hammer and Stanley B. Zdonik. Knowledge-based query processing. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, pages 137–147. VLDB Endowment, 1980.

[122] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, SIGMOD '04, pages 647–658, New York, NY, USA, 2004. ACM.

[123] Yuchen Liu, Hai Liu, Dongqing Xiao, and Mohamed Y. Eltabakh. Adaptive correlation exploitation in big data query optimization. *The VLDB Journal*, 27(6):873–898, Dec 2018.

[124] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. *SIGMOD Rec.*, 29(2):105–116, May 2000.

[125] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of two semantic query optimization techniques in db2 universal database. In *25th VLDB*, VLDB '99, pages 687–698, 1999.

[126] Ziheng Wei and Sebastian Link. Embedded functional dependencies and data-completeness tailored database design. *PVLDB*, 12(11):1458–1470, 2019.

[127] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. pages 164–175, 2014.

[128] Wenfei Fan, Floris Geerts, and Jef Wijsen. Determining the currency of data. *ACM Transactions on Database Systems*, 37(4):25:1–25:46, 2012.

[129] Noël Novelli and Rosine Cicchetti. FUN: An efficient algorithm for mining functional and embedded dependencies. pages 189–203, 2001.

[130] Eduardo H. M. Pena and Eduardo Cunha de Almeida. BFASTDC: A bitwise algorithm for mining denial constraints. pages 53–68, 2018.

[131] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[132] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, 2003.

[133] Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. The parameterized complexity of dependency detection in relational databases. In *International Symposium on Parameterized and Exact Computation (IPEC)*, pages 6:1–6:13, 2016.

[134] Li Lin and Yunfei Jiang. The computation of hitting sets: Review and new algorithms. *Information Processing Letters*, 86(4):177 – 184, 2003.

[135] J. Bailey, T. Manoukian, and Kotagiri Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. pages 485–488, 2003.

[136] Thorsten Papenbrock, Tanja Bergmann, Moritz Finke, Jakob Zwiener, and Felix Naumann. Data profiling with Metanome. *PVLDB*, 8(12):1860–1863, 2015.

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

[138] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proc. VLDB Endow.*, 9(12):993–1004, August 2016.

[139] Charles Truong, Laurent Oudre, and Nicolas Vayatis. Selective review of offline change point detection methods. *Signal Processing*, 167:107299, 2020.

[140] R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.

[141] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Lightning fast and space efficient inequality joins. 8(13):2074–2085, 2015.

[142] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. page 10–20, 1999.

[143] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exper.*, 46(11):1547–1569, 2016.

[144] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD Rec.*, 22(2):267–276, 1993.

[145] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. 27(5):643–668, 2018.

[146] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Correlation maps: A compressed access method for exploiting soft functional dependencies. *Proc. VLDB Endow.*, 2(1):1222–1233, August 2009.

[147] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315:972–976, 2007.

[148] Jayanta Basak, Kushal Wadhwani, and Kaladhar Voruganti. Storage workload identification. *Trans. Storage*, 12(3):14:1–14:30, May 2016.

[149] M. Hazewinkel. *Encyclopaedia of Mathematics (1)*. Springer, 1987.

[150] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. *SIGMOD Rec.*, 25(2):57–67, June 1996.

[151] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, June 1990.

[152] D. Laurent and N. Spyratos. Rewriting aggregate queries using functional dependencies. In *International Conference on Management of Emergent Digital EcoSystems*, pages 40–47, New York, NY, USA, 2011. ACM.

[153] G. N. Paulley and Per-Ake Larson. Exploiting uniqueness in query optimization. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 127–145, Riverton, NJ, USA, 2010. IBM Corp.

[154] Shashi Shekhar, Jaideep Srivastava, and Soumitra Dutta. A formal model of trade-off between optimization and execution costs in semantic query optimization. In *14th VLDB*, VLDB '88, pages 457–467, San Francisco, CA, USA, 1988.

[155] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD 2008, Vancouver, BC, Canada*, pages 967–980, 2008.

[156] Cleide L. B. Possamai, Diego Pasqualin, Daniel Weingaertner, Eduardo Todt, Marcos A. Castilho, Luis C. E. de Bona, and Eduardo Cunha de Almeida. Proinfodata: Monitoring a large park of computational laboratories. In Luis Corral, Alberto Sillitti, Giancarlo Succi, Jelena Vlasenko, and Anthony I. Wasserman, editors, *Open Source Software: Mobile Open Source Technologies*, pages 226–229, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[157] Sumita Barahmand and Shahram Ghandeharizadeh. D-zipfian: A decentralized implementation of zipfian. In *Proceedings of the Sixth International Workshop on Testing Database Systems*, DBTest '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[158] Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. Approximate discovery of functional dependencies for large datasets. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, page 1803–1812, New York, NY, USA, 2016. Association for Computing Machinery.

[159] Sebastian Kruse, Thorsten Papenbrock, Christian Dullweber, Moritz Finke, Manuel Hegner, Martin Zabel, Christian Zoollner, and Felix Naumann. Fast approximate discovery of inclusion dependencies. In *Datenbanksysteme fur Business, Technologie und Web (BTW 2017)*, pages 207–226. Gesellschaft fur Informatik, Bonn, 2017.

[160] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. Approximate denial constraints. *Proc. VLDB Endow.*, 13(10):1682–1695, 2020.

[161] Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Detecting unique column combinations on dynamic data. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1036–1047. IEEE Computer Society, 2014.

[162] Loredana Caruccio and Stefano Cirillo. Incremental discovery of imprecise functional dependencies. *J. Data and Information Quality*, 0(ja).

APPENDIX A – PUBLICATIONS

1. **Eduardo H. M. Pena** and Eduardo Cunha de Almeida. Short paper: Uso de instâncias de dados e carga de trabalho para mineração de restrições de integridade. Brazilian Symposium on Databases (SBBD), pages 312–317, 2017.

2. **Eduardo H. M. Pena**. Workload-Aware Discovery of Integrity Constraints for Data Cleaning. PhD Workshop co-located with the 44th International Conference on Very Large Databases (VLDB), 2018.

3. **Eduardo H. M. Pena**, Erik Falk, Jorge Augusto Meira, and Eduardo Cunha de Almeida. Mind Your Dependencies for Semantic Query Optimization. Journal of Information and Data Management (JIDM), pages 3–19, 2018.

4. **Eduardo H. M. Pena** and Eduardo Cunha de Almeida. BFASTDC: A Bitwise Algorithm for Mining Denial Constraints. Database and Expert Systems Applications (DEXA), pages 53–68, 2018.

5. **Eduardo H. M. Pena** and Eduardo Cunha de Almeida. Short paper: Descoberta automática de restrições de negação confiáveis. Brazilian Symposium on Databases (SBBD), pages 187–192, 2019.

6. **Eduardo H. M. Pena**, Eduardo Cunha de Almeida, and Felix Naumann. Discovery of Approximate (and Exact) Denial Constraints. Proc. VLDB Endow. (PVLDB), pages 266–278, 2019.

7. **Eduardo H. M. Pena**, Edson R. Lucas Filho, Eduardo Cunha de Almeida, and Felix Naumann. Efficient Detection of Data Dependency Violations. To appear in Proceedings of the 29th ACM International on Conference on Information and Knowledge Management (CIKM), 2020.

8. Fabiola Santore, Eduardo Cunha De Almeida, Wagner H. Bonat, **Eduardo H. M. Pena**, Luiz Eduardo S. Oliveira. A Framework for Analyzing the Impact of Missing Data in Predictive Models. To appear in Proceedings of the 29th ACM International on Conference on Information and Knowledge Management (CIKM), 2020.