AN ABSTRACT OF THE THESIS OF

Ga Young Lee for the degree of Master of Science in Computer Science presented on December 17, 2021.

Title: Relational Learning over Dirty Data Using Data Constraints

Abstract approved: ______________________________

Arash Termehchy

Real-world datasets are dirty and contain many errors. Examples of these issues are violations of integrity constraints, duplicates, and inconsistencies in representing data values and entities. Applying machine learning on dirty databases may lead to inaccurate results. Users have to spend a great deal of time and effort repairing data errors and creating a clean learning database. Moreover, as the information required to fix these errors is not often available, there may be numerous possible clean versions for a dirty database. We propose *DLearn*, a novel relational learning system that learns directly over dirty databases effectively and efficiently without any preprocessing. *DLearn* leverages database constraints, such as functional dependency and matching dependency, to learn accurate relational models over inconsistent and heterogeneous data. Its learned models using the unique data properties represent patterns over all possible clean instances of the data in a usable form. Our empirical study indicates that *DLearn* learns accurate models over large real-world databases efficiently.

©Copyright by Ga Young Lee
December 17, 2021
All Rights Reserved

Relational Learning over Dirty Data Using Data Constraints

by

Ga Young Lee

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 17, 2021
Commencement June 2022

Master of Science thesis of Ga Young Lee presented on December 17, 2021.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Ga Young Lee, Author

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Professor Arash Termehchy, who took a chance on me and provided unlimited support and guidance to help me become a researcher. Since I joined the IDEA Lab in 2019, I have met brilliant people I'm proud to call my colleagues and friends, including my collaborator Jack Davis. Through our shared passion for data and technology, I had eye-opening discussions about the use of technology in society and our roles as engineers and scientists. I am beyond grateful to everyone who challenged my viewpoint and generously offered their time and advice, including my committee members, Professor Liang Huang, Professor Weng-Keen Wong, and Professor Ren Guo. Lastly, my family members in Seoul and Chicago and my partner, Matthew Castillon, never stopped supporting me and believing in me throughout my graduate studies. I hope to give back the love I have received during the most challenging yet rewarding years of my life.

CONTRIBUTION OF AUTHORS

Dr. Arash Termehchy primarily led this research project as an extension of Dr. Jose Picado's thesis. Jack Davis and I refined the bottom-clause construction algorithm, ran the experiments to collect empirical data, and analyzed the implications. Specifically, I played a key role in creating a noise injection algorithm to test the model's performance and analyzed the empirical data for its implications against benchmarks. Also, I worked on refining the bottom-clause construction algorithms and communicated our contributions in manuscripts and presentation materials to help the broader audience understand the novel data cleaning algorithm.

TABLE OF CONTENTS

Page

1 Introduction 1

2 Literature Review 6

2.1 Related Work 6

2.2 Open Problem 9

2.2.1 Lack of Optimizer 9

2.2.2 Tradeoff Between Efficiency and Coverage 9

2.2.3 Limited Generalizability 10

3 Preliminary 12

3.1 Relational Learning 12

3.2 Matching Dependencies 14

3.3 Conditional Functional Dependencies 17

3.4 Semantic of Learning 18

3.4.1 Different Approaches 18

3.4.2 Heterogeneity in Definitions 20

3.4.3 Coverage over Heterogeneous Data 23

4 Methods 26

4.1 DLearn 26

4.1.1 Bottom-Clause Construction 26

4.1.2 Generalization 30

4.1.3 Efficient Coverage Testing 33

4.1.4 Commutativity of Cleaning and Learning 34

4.2 Implementation 35

5 Experiments 37

5.1 Experimental Settings 37

5.1.1 Datasets 37

5.1.2 CFDs 38

5.1.3 Systems, Metrics, and Environment 39

5.2 Empirical Results 40

5.2.1 Handling MDs 40

5.2.2 Handling MDs and CFDs 42

5.2.3 Impact of Number of Iterations 43

TABLE OF CONTENTS (Continued)

Page

5.2.4 Scalability of DLearn 43
5.2.5 Effect of Sampling 44

6 Conclusion and Discussion 46
6.1 Summary 46
6.2 Limitations 46
6.3 Future Research Directions 46

Bibliography 46

LIST OF FIGURES

Figure Page

5.1 Learning over the IMDb+OMDb (3 MDs) dataset while increasing the number of positive and negative (#P, #N) examples (left) and while increasing sample size for $k_m = 2$ (middle) and $k_m = 5$ (right). 44

LIST OF TABLES

Table | Page

1.1 Schema fragments for the IMDb and BOM. 2

4.1 Example movie database 27

5.1 Numbers of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset. 37

5.2 Results of learning over all datasets with MDs. Number of top similar matches denoted by k_m. 40

5.3 Results of learning over all datasets with MDs and CFD violations. p is the percentage of CFD violation. 43

5.4 Results of changing the number of iterations. 43

5.5 Learning over the IMDb+OMDb (3 MDs) with CFD violations by increasing positive (#P) and negative (#N) examples. 44

LIST OF ALGORITHMS

Algorithm — Page

1 Covering approach algorithm. 13

2 DLearn bottom-clause construction algorithm. 28

Chapter 1: Introduction

Dirty data is a persistent problem in data collection processes; as the amount of available data grows, the likelihood that errors and noises will contaminate the data also increases. Using degraded data in training machine learning models can result in a considerable loss in performance [52]. Hence, it is highly advisable to inspect the data and ensure its quality by removing dirty data, such as null values, bogus tuples, and logical inconsistency. However, the manual review and cleaning process can be time-consuming due to the sheer volume of the data that must be processed for training models. This thesis explores currently available data-cleaning methods; suggests a novel approach that leverages a unique characteristic of data, namely entity matching; and validates the effectiveness of this approach in data cleaning.

Users often want to learn interesting relationships over relational databases [55]. Consider the IMDb database (imdb.com), which contains information about movies whose schema fragments are shown in Table 1.1. Given a relational database and training examples for a new relation, *relational machine learning* (relational learning) algorithms learn (approximate) relational models and definitions of the target relation in terms of existing relations in the database [17, 29, 58, 48, 51, 54]. For instance, the user may provide a set of high grossing movies as positive examples and a set of low grossing movies as negative examples to a relational learning algorithm. Given the IMDb database and these examples, the algorithm may learn:

$$\begin{aligned} \mathit{highGrossing}(x) \leftarrow & \mathit{movies}(y, x, z), \mathit{mov2genres}(y, \text{`}\mathit{comedy}\text{'}), \\ & \mathit{mov2releasedate}(y, \text{`}\mathit{May}\text{'}, u), \end{aligned}$$

which indicates that high grossing movies are often released in May and fall into the *comedy* genre. One may assign weights to these definitions to describe their prevalence in the data according to their training accuracy [39, 58]. Unlike other machine learning algorithms, relational learning methods do not require the data points to be statistically independent and follow the same identical distribution (IID) [19]. Since a relational

IMDb	
movies(id, title, year)	mov2countries(id, name)
mov2genres(id, name)	mov2releasedate(id, month, year)
BOM	
mov2totalGross(title, gross)	
highBudgetMovies(title)	

Table 1.1: Schema fragments for the IMDb and BOM.

database usually contains information about multiple types of entities, the relationships between these entities often violate the IID assumption. Moreover, the data about each type of entity may follow a distinct distribution. This also holds if one wants to learn over the data gathered from multiple data sources, as each data source may have a distinct data distribution. Thus, applying other learning methods to these databases can result in biased and inaccurate models [39, 56, 19]. Since relational learning algorithms leverage the structure of a database directly to learn new relations, they do not reply on the tedious process of feature engineering. In fact, they are used to identify features for the downstream non-relational models [45]. Thus, they have been widely used over relational data, for example, building usable query interfaces [3, 47, 35], information extraction [39, 19], and entity resolution [21].

Real-world databases often contain inconsistencies [10, 18, 24, 28, 15, 61, 23], which may prevent the relational learning algorithms from finding an accurate definition. In particular, the information in a domain is sometimes spread across several databases. For example, IMDb does *not* contain the information about the budgets or total grosses of movies. This information is available in another database called Box Office Mojo (BOM) (*boxofficemojo.com*), schema fragments for which are shown in Table 1.1. To determine an accurate definition for *highGrossing*, the user must collect data from the BOM database. However, the same entity or value may be represented in various forms in the original databases. For example, the *titles* of the same movie in IMDb and BOM have different formats (e.g., the title of the movie *Star Wars: Episode IV* is represented in IMDb as *Star Wars: Episode IV - 1977* and in BOM as *Star Wars - IV*).

A single database may also contain these types of heterogeneity, as a relation may have duplicate tuples for the same entity; for example, there may be duplicate tuples for the same movie in BOM. A database may have other types of inconsistencies that

violate the integrity of the data. For example, a movie listed on IMDb may have two different production years [15, 61, 23].

Users have to resolve inconsistencies and learn over the repaired database, which is exceedingly difficult and time-consuming when it comes to large databases [18, 28]. First, the user must develop or train a matching function that distinguishes and unifies different values that refer to the same entity, applies that function to the database, and materializes the produced instance. Second, a unification may lead to new inconsistencies and opportunities to do more cleaning. For example, after unifying the titles of movies in a database, the user may notice that the names of directors of these movies are different in the database. To keep the database consistent, the user has to unify the names of the directors whose movies have been unified in the recently produced database. This process will be repeated for entities related to directors, e.g., production companies with which a director has worked and other entities related to the movies until there are no more values to reconcile. Thus, it will take a long time and extensive manual work to produce and materialize a clean database instance.

Repairing inconsistencies usually leads to numerous clean instances, as information about the correct fixes is seldom available [10, 13, 24]. An entity may match and be a potential duplicate of multiple distinct entities in the database. For example, the title *Star Wars* may match both the titles *Star Wars: Episode IV - 1977* and *Star Wars: Episode III - 2005*. Since we know that the *Star Wars: Episode IV - 1977* and *Star Wars: Episode III - 2005* refer to two different movies, the title *Star Wars* must be linked to only one of them. For each choice, the user ends up with a distinct database instance. Since a large database may have many potential matches, the number of clean database instances will extremely high.

Similarly, it is often not clear how data integrity violations should be resolved. For instance, if a movie has multiple production years, one may not know which year is correct. Due to the sheer number of volume of data, it is not possible to generate and materialize all clean instances for a large dirty database [23]. Cleaning systems usually produce a subset of all clean instances, for example, those that differ minimally from the original data [23]. This approach still generates many repaired databases [10, 61, 23]. It has been also shown that these conditions may not produce the correct instances [34]. Thus, the cleaning process may result in many instances where it is not clear which one should be used for learning. Sine users do not often have the required information to

find the right fixes, they may not end up with the correct instance and effective model. As a result, most data scientists spend more than 80% of their time on such cleaning tasks [43].

Some systems aim to produce a single probabilistic database that contains information about a subset of possible clean instances [57]. These systems, however, do not address the problem of duplicates and value heterogeneities, as they assume that there always is a reliable table, akin to a dictionary, which provides the unique value that should replace each potential duplicate in the database. However, given that different values represent the same entity, it is *not* clear what should replace the final value in the clean database (e.g., whether *Star War* represents *Star Wars: Episode IV - 1977* or *Star Wars: Episode III - 2005*). Such systems also allow violations of integrity constraints to generate the final probabilistic database efficiently, which may lead to inconsistent repairs. Moreover, to restrict the set of clean instances, attributes must have finite domains, which does not generally hold in practice.

We propose a novel learning method that learns directly over dirty databases without materializing their clean versions, which substantially reduces the effort needed to learn over dirty [52]. The properties of clean data are usually expressed using declarative data constraints, for example, functional dependencies, [2, 1, 14, 22, 24, 7, 23, 13, 57, 26]. Our system uses the declarative constraints during learning. These constraints may be provided by users or discovered from the data using profiling techniques [1, 41]. Our contributions are as follows:

- We introduce and formalize the problem of learning over an inconsistent database (Section 3.4).
- We propose a novel relational learning algorithm called *DLearn* to learn over inconsistent data (Section 4.1).
- Every learning algorithm chooses the final result based on its coverage of the training data.
- We propose an efficient method for computing the coverage of a definition directly over the heterogeneous database (Section 4.1.2).
- We provide an efficient implementation of DLearn over a relational database system (Section 4.2).

- We conduct an extensive empirical study over real-world datasets and show that DLearn scales to and learns efficiently and effectively over large data.

The proof of our theoretical results is presented in [52].

Chapter 2: Literature Review

2.1 Related Work

Data cleaning is an important and flourishing area in database research [24, 14, 7, 23, 13]. In the traditional data cleaning, practitioners write regular expressions to eliminate potentially incorrect or missing values from training data, which can be enhanced by the human-in-the-loop interface [46]. However, the rule-based approach can be expensive, time-consuming, and, most importantly, inaccurate. Several data cleaning papers focus on updating tuples that violate the data integrity constraints by leveraging declarative constraints to address this problem [14, 24, 7, 23, 13, 57, 43]. On the other hand, others combine the instances and represent them as a probabilistic database [57]. In this section, we analyze a body of related research in the field and describe the motivation for our approach to directly learn over the original data instead of materializing its repairs.

SampleClean: Simulated Clean Data Instances

SampleClean suggests a solution to sample the raw data that can better present clean data instances. A naive sampling approach can be misleading because the semantics of dirty data differ from those of clean data, so the researchers used approximate query processing (AQP) to solve this potential issue [42]. AQP consists of two steps: first, in the direct estimate (DE) stage, a set of k rows is sampled randomly and cleaned, and the training result is returned independently of the dirty data. However, DE alone can lead to incorrect answers when the dirty data is dominant in the sampled subset. To mitigate this drawback, a correction step is used to reweight the sample based on the contribution of the cleaned data to the entire dataset when it is used in training. By adopting this framework, Krishnan et al. were able to reach an equilibrium where the approximate query result is bounded by the confidence interval that can ensure the proper amount of clean data is captured in the simulation. To do so, the authors adopted a probabilistic model to reweight samples based on their contribution to the

entire dataset. In addition, a soft boundary was imposed to ensure the proper amount of data was captured in the simulated sample. For example, suppose a user wants to query data to obtain a certain amount of accurate data. In that case, SampleClean suggests the sample size recommended to meet the desired level of accuracy, which is bounded asymptotically by estimators. Moreover, by leveraging the stochastic gradient descent, the average value of transformed data is calculated and converges at the optimal value. However, the simulation of cleaned data instances is insufficient for identifying the correct version of clean data. To address this issue, the same group of researchers published a follow-up study on the data cleaning method called ActiveClean.

ActiveClean: Incremental Data Cleaning in Convex Models

Unlike SampleClean, which involves constructing a simulated clean data instance, the researchers suggested iteration while retaining the models based on the optimizer. The optimizer is a stochastic gradient descent that iteratively samples data, estimates a gradient, and updates the current best model. By applying this method, practitioners only need to retrain with a small subset instead of the entire dataset [43]. The objective of ActiveClean is to learn a model over dirty data without cleaning and transforming the data. ActiveClean gradually cleans a dirty dataset to learn a convex-loss model, such as logistic regression and support vector machine (SVM). The key difference is that ActiveClean aims to clean the underlying dataset incrementally so that the learned model becomes more effective as it receives more cleaned records. However, this approach has some drawbacks. For example, the applicability of ActiveClean is limited to convex models, and it requires user input, such as the specification of the model, gradient, and stopping criteria.

HoloClean: Data Repair with Probabilistic Inference

Improvements in probabilistic inference enabled the creation of another framework for data cleaning [57]. The main difference compared to other approaches in the field is that this framework considers marginal probability when given the candidate sample to clean. The statistics and the dataset provided by users help the model to reason about the nature of dirty data. As an initial approach, the authors developed a model that

learns on both clean and corrupted data to infer where potential errors are highly likely to occur. The authors propose training a model on a relatively small portion of real data and checking whether the model can infer the nature of the mistakes made by humans when filling out the data. When the authors evaluated the model on a whole dataset, the accuracy of the test results was over 90%, which indicated that the proposed inference model is promising. HoloClean needs both clean and dirty samples to train an optimal model, requiring cleaning and verifying the training set. However, it is difficult to obtain sufficient clean and dirty data samples to learn from.

AlphaClean: Generate-Then-Search Parallel Data Cleaning

The authors of AlphaClean propose a system that would automate the process of finding an effective pipeline of cleaning tools [44]. The process of building a candidate pipeline includes parameter selection of the specific pipeline to be generated. The AlphaClean solution works with any given number of cleaning tools, which users work with to clean the data. However, the number of given tools should be reasonably limited because of the way in which the algorithm generates the cleaning pipeline. It builds a combination of pipelines in a tree-shaped manner. As the algorithm progresses, the solution becomes available progressively, and the output is revised in further iterations, which aims to improve the quality of the proposed pipeline. The pipelines can be accessed and evaluated by the user at any given time while the algorithm is running. This framework offers the flexibility to assess the outcome from each step, allowing users to check whether the proposed branch of data cleaning pipelines is promising and should be considered as a candidate solution.

CPClean: Reusable Computation in Data Cleaning

In a paper titled "Nearest Neighbor Classifiers over Incomplete Information" [36], the authors propose several solutions for the inconsistencies that are assumed to impact machine learning. The solutions include using checking and counting as means by which to study the impact of incomplete data on training machine learning applications. The paper also suggests an extensive data collection for the CPClean approach, which was developed based on certain predictions (CP) primitives, the performance of which has

shown promise on datasets with missing values. When applied on five datasets with missing values, CPClean closes a 100% gap on average by cleaning up to 36% of dirty data. In contrast, other data collecting approaches, such as BoostClean, the best automatic cleaning approach, can only clean a 14% gap on average. CPClean has an advantage over other cleaning approaches as it closes 100% of all cases, whereas other techniques, such as BoostClean, fail to achieve satisfactory performance [36]. However, the available classifier in this method is limited to the nearest-neighbor classifier. Therefore, the CPClean has a critical limitation as it lacks extensibility.

2.2 Open Problem

2.2.1 Lack of Optimizer

Current data cleaning solutions lack a clear interface that suggests the optimal stopping point for the amount of data to be cleaned. As a result, practitioners rarely reach the equilibrium required to ensure a desirable level of clean data in a training set. ActiveClean aims to address this problem by allowing users to set the parameters, such as models, gradient, and stopping criteria [43]. However, these manual manipulations require domain expertise and a deep understanding of the parameter tuning technique. Therefore, the applicability of ActiveClean's optimizer is limited to a small subset of users who can benefit from data cleaning. In other words, a limitation of the ActiveClean model is that the user must have a certain level of technical knowledge to manipulate the model and troubleshoot by opening the diagnostic panel when it is making poor predictions. Furthermore, the model does not show where the cleaned data is stored or whether it will ever be recovered. To overcome this challenge, data-checking models can be designed based on the data problems to be addressed and user preferences. Therefore, it is suggested that a future data cleaning solution should consider the user needs and problems to be solved and communicate its progress throughout the process.

2.2.2 Tradeoff Between Efficiency and Coverage

Earlier data cleaning techniques, such as SampleClean and HoloClean, focus on the extensive data cleaning that covers the entire data instances, which can guarantee a

high level of coverage at the price of significant processing time [42, 57]. In contrast, the more recent approaches based on inference and logic are promising in using little data to generate clean instances. However, due to the sparsity observed in raw data, this approach of using inference and learning does not ensure optimal coverage of the data. This tradeoff between efficiency and coverage remains a perennial problem in data cleaning that must be addressed by designing an efficient algorithm or using caches to store the preprocessed subset of raw data. The research suggests that data cleaning is becoming less costly and capable of covering more data with the advent of widely available high-performing computing resources [52]. Despite the progress, generating cleaning language for various systems is still challenging due to the different modes and procedures of data productions. Specifically, while the repair functions are complex, they sometimes do not function as intended. Even though parallelism is a beneficial tool for enhancing performance, the data cleaning system is prone to repair each instance, resulting in redundancy. On the other hand, the sequential data cleaning process cannot perform complex tasks in different databases. Therefore, these systems are not effective for a large-scale database solution. In the case of AlphaClean [44], the authors suggest modifications to increase the flexibility of the data cleaning solution and extend it to be integrated with visualization tools, which would enhance the usability of the system and allow it to adapt to a broader range of needs.

2.2.3 Limited Generalizability

The empirical results concerning the data cleaning methods indicate that some methods are hyper-specific to certain data types. To expand the generalizablity of a system, it is crucial to design and test a data cleaning method that can learn from a given dataset. This problem is most notable for CPClean. Specifically, the limitation of the "Nearest Neighbor Classifiers over Incomplete Information" paper is that the authors performed their experiments on only five datasets [36]; this number is arbitrary and difficult to justify for other applications. There is a high likelihood that the method may not work well with different experiment settings. Another limitation is that it is unclear whether their approach will work with other classifiers or just the K-nearest neighbor (KNN) classifier. While their approach using KNN offered linear complexity, they did not explain the reasoning behind the choice of KNN beyond the simplicity of

the algorithm. The future work for this paper would be to explore the effectiveness of other classifiers and determine if it would be possible to reach similar time complexity as that achieved using KNN.

Chapter 3: Preliminary

3.1 Relational Learning

This section reviews the basic concepts of relational learning over databases without any heterogeneity [17, 29]. We fix two mutually exclusive sets of relation and attribute symbols. A database schema $\mathcal{S}$ is a finite set of relation symbols R_i, $1 \leq i \leq n$. Each relation R_i is associated with a set of attribute symbols denoted as $R_i(A_1, \ldots, A_m)$. We denote the domain of values for attribute A as $dom(A)$. Each database instance I of schema $\mathcal{S}$ maps a finite set of tuples to every relation R_i in $\mathcal{S}$. Each tuple t is a function that maps each attribute symbol in R_i to a value from its domain. We denote the value of the set of attributes X of tuple t in the database I by $t^I[X]$ or $t[X]$ if I is clear from the context. Moreover, when it is clear from the context, we refer to an instance of a relation R simply as R. An *atom* is a formula in the form of $R(u_1, \ldots, u_n)$, where R is a relation symbol and $u_1, \ldots, u_n$ are *terms*. Each term is either a variable or a constant, i.e., value. A *ground atom* is an atom that only contains constants. A *literal* is an atom, or the negation of an atom. A *Horn clause* (clause for short) is a finite set of literals that contains exactly one positive literal. A *ground clause* is a clause that only contains ground atoms. Horn clauses are also called Datalog rules (without negation) or conjunctive queries. A *Horn definition* is a set of Horn clauses with the same positive literal, i.e., non-recursive Datalog program or union of conjunctive queries. Each literal in the body is *head-connected* if it has a variable shared with the head literal or another head-connected literal.

Relational learning algorithms learn first-order logic definitions from an input relational database and training examples. Training examples E are usually tuples of a single *target relation*, and express positive (E^+) or negative (E^-) examples. The input relational database is also called *background knowledge*. The *hypothesis space* is the set of all possible first-order logic definitions that the algorithm can explore. It is usually restricted to Horn definitions to keep learning efficient. Each member of the hypothesis space is a *hypothesis*. Clause C *covers* an example e if $I \wedge C \models e$, where $\models$ is the entail-

ment operator: in other words, if I and C are true, then e is true. Definition H covers an example e if at least one of its clauses covers e. The goal of a learning algorithm is to find the definition in the hypothesis space that covers all positive and the fewest negative examples as possible.

Example 3.1.1. *IMDb contains the tuples movie (10, 'Star Wars: Episode IV - 1977', 1977), mov2genres(10, 'comedy'), and mov2releasedate(10, 'May', 1977). Therefore, the definition that indicates that high grossing movies are often released in May and fall in the comedy genre is shown in Section 1 covers the positive example highGrossing('Star Wars: Episode IV - 1977').*

Most relational learning algorithms follow a covering approach, as illustrated in Algorithm 1 [48, 51, 53, 54, 62]. The algorithm constructs one clause at a time using the *LearnClause* function. If the clause satisfies a criterion, (e.g., it covers at least a certain fraction of the positive examples and does *not* cover more than a certain fraction of negative ones), the algorithm adds the clause to the learned definition and discards the positive examples covered by the clause. It stops when all positive examples are covered by the learned definition. For example, the *LearnClause* function in Algorithm 1 usually computes the difference of the number of positive and negative examples as the score and picks the clause(s) with the highest score.

Algorithm 1: Covering approach algorithm.

```
Input  : Database instance I, examples E
Output: Horn definition H
1 H = {}
2 U = E+
3 while U is not empty do
4     C = LearnClause(I, U, E−)
5     if C satisfies minimum criterion then
6         H = H ∪ C
7         U = U − {e ∈ U | H ∧ I ⊨ e}
8 return H
```

3.2 Matching Dependencies

Learning over databases with heterogeneity in representing values may deliver inaccurate answers as the same entities and values may be represented under different names. Thus, one must resolve these representational differences to produce a high-quality database to learn an effective definition.

Suppose one wants to match and resolve values in a couple of attributes. In that case, one may use a supervised or unsupervised matching function to identify and unify their potential matches according to the domain of those attributes. Users may apply string similarity functions, such as edit distance, to find potential matches if the domain of attributes is a set of strings. Also, matching and resolution rules can be used to find a match based on domain knowledge. For example, consider relation *Employee(id, name, home-phone, address)*. It can be inferred that if the phone numbers of two tuples are sufficiently similar, e.g., *001-333-1020* and *333-1020*, then their addresses must be equal. Knowing this rule, the user may manipulate the database to ensure that all tuples with a similar phone number have equal addresses.

There may be a multiple matching rules in a large relational database and they may interact with each other [5, 7, 9, 13, 24, 26, 33, 32, 59]. For example, given the relation *Employee(id, name, home-phone, address)*, it is known that if two tuples have sufficiently similar addresses, e.g., *1 Main St., NY* and *1 Main Street, New York*, and names, they must have the same values for attribute *id*. Now, consider two tuples whose names and home phone numbers are sufficiently similar, but their addresses are *not*. According to this rule, one *cannot* unify the ids of these tuples. But, after applying the rule mentioned in the preceding paragraph on the phone number and address, their addresses become sufficiently similar. Then, one can use the second rule to unify the values of *id* in these tuples.

The database community has proposed declarative matching and resolution rules to express the domain knowledge about matching and resolution [5, 7, 9, 13, 24, 26, 33, 32, 59]. *Matching dependencies (MD)* are a popular type of such declarative rules, which provide a powerful method of expressing domain knowledge on matching values [24, 10, 8, 23, 41].

Let $\mathcal{S}$ be the schema of the original database and R_1 and R_2 two distinct relations in $\mathcal{S}$. Attributes A_1 and A_2 from relations R_1 and R_2, respectively, are comparable if they

share the same domain. MD σ is a sentence of the form $R_1[A_1] \approx_{dom(A_1)} R_2[B_1], \ldots, R_1[A_n] \approx_{dom(A_n)} R_2[B_n] \rightarrow R_1[C_1] \rightleftharpoons R_2[D_1], \ldots, R_1[C_m] \rightleftharpoons R_2[D_m]$, where A_i and C_j are comparable to B_i and D_j, respectively, $1 \leq i \leq n$, and $1 \leq j \leq m$. Operation $\approx_d$ is a similarity operator defined over domain d and $R_1[C_j] \rightleftharpoons R_2[D_j]$,$1 \leq j \leq m$, indicates that the values of $R_1[C_j]$ and $R_2[D_j]$ refer to the same value (i.e., they are interchangeable). Intuitively, the aforementioned MD says that if the values of $R_1[A_i]$ and $R_2[B_i]$ are sufficiently similar, the values of $R_1[C_j]$ and $R_2[D_j]$ are different representations of the same value.

For example, consider again the database that contains relations from *IMDb* and *BOM* the schema fragments of which are shown in Table 1.1. According to the discussion in Section 1, one can define the following MD σ_1 : *movies[title]* $\approx$ *highBudgetMovies[title]* $\rightarrow$ *movies[title]* $\rightleftharpoons$ *highBudgetMovies[title]*. The exact implementation of the similarity operator depends on the underlying domains of attributes. Our results are orthogonal to the implementation details of the similarity operator.

In the remainder of this thesis, we use $\approx_d$ operation only between comparable attributes. For brevity, we eliminate the domain d from $\approx_d$ when it is clear from the context or the results hold for any domain d. We also denote $R_1[A_1] \approx R_2[B_1], \ldots, R_1[A_n] \approx R_2[B_n]$ in an MD as $R_1[A_{1\ldots n}] \approx R_2[B_{1\ldots n}]$. An MD $R_1[A_{1\ldots n}] \approx R_2[B_{1\ldots n}] \rightarrow R_1[C_1] \rightleftharpoons R_2[D_1], \ldots, R_1[C_m] \rightleftharpoons R_2[D_m]$ is equivalent to a set of MDs $R_1[A_{1\ldots n}] \approx R_2[B_{1\ldots n}] \rightarrow R_1[C_1] \rightleftharpoons R_2[D_1]$, $R_1[A_{1\ldots n}] \approx R_2[B_{1\ldots n}] \rightarrow R_1[C_2] \rightleftharpoons R_2[D_2], \ldots, \rightarrow R_1[C_1] \rightleftharpoons R_2[D_1] \rightarrow \ldots, R_1[C_m] \rightleftharpoons R_2[D_m]$. Thus, for the rest of the thesis, we assume that each MD is in the form of $R_1[A_{1\ldots n}] \approx R_2[B_{1\ldots n}] \rightarrow R_1[C] \rightleftharpoons R_2[D]$, where C and D are comparable attributes of R_1 and R_2, respectively.

Given a database with MDs, one must enforce the MDs to generate a high-quality database. Let tuples t_1 and t_2 belong to R_1 and R_2 in database I of schema $\mathcal{S}$, respectively, such that $t_1^I[A_i] \approx t_2^I[B_i]$, $1 \leq i \leq n$, denoted as $t_1^I[A_{1\ldots n}] \approx t_2^I[B_{1\ldots n}]$ for brevity. To enforce the MD $\sigma : R_1[A_{1\ldots n}] \approx R_2[B_{1\ldots n}] \rightarrow R_1[C] \rightleftharpoons R_2[D]$ on I, one must make the values of $t_1^I[C]$ and $t_2^I[D]$ identical as they actually refer to the same value [24, 10]. For example, if attributes C and D contain titles of movies, one unifies both values *Star Wars - 1977* and *Star Wars - IV* to *Star Wars Episode IV - 1977* as it deems this value as the one to which $t_1^I[C]$ and $t_2^I[D]$ refer. The following definition formalizes the concept of applying an MD to the tuples t_1 and t_2 on I.

Definition 3.2.1. *Database I' is the immediate result of enforcing MD σ on t_1 and t_2 in I, denoted by $(I, I')_{[t_1,t_2]} \models \sigma$ if*

1. $t_1^I[A_{1...n}] \approx t_2^I[B_{1...n}]$, *but* $t_1^I[C] \neq t_2^I[D]$*;*

2. $t_1^{I'}[C] = t_2^{I'}[D]$ *; and*

3. *I and I' agree on every other tuple and attribute value.*

One may define a unification function over some domains to map the values that refer to the same value to the correct value in the cleaned instance. It is, however, usually difficult to define such a function due to the lack of knowledge about the correct value. For example, let C and D in Definition 3.2.1 contain information about names of people and $t_1^I[C]$ and $t_2^I[D]$ have the values *J. Smth* and *Jn Sm*, respectively, which, according to an MD, refer to the same actual name, which is *Jon Smith*. It is not clear how to compute *Jon Smith* using the values of $t_1^I[C]$ and $t_2^I[D]$. We know that the values of $t_1^{I'}[C]$ and $t_2^{I'}[D]$ will be identical after enforcing σ, but we usually do not know their exact values. Because we aim to develop learning algorithms that are efficient and effective over databases from various domains, we do *not* fix any matching method in this thesis. We assume that matching every pair of values a and b in the database creates a fresh value denoted as $v_{a,b}$.

Given the database I with the set of MDs Σ, I' is *stable* if $(I, I')_{[t_1,t_2]} \models \sigma$ for all $\sigma \in \Sigma$ and all tuples $t_1, t_2 \in I'$. In a stable database instance, all values that represent the same data item according to the database MDs are assigned equal values. Thus, it does not have any heterogeneities. Given a database I with set of MDs Σ, one can produce a stable instance for I by starting from I and iteratively applying each MD in Σ according to Definition 3.2.1 finitely many times [24, 10]. Let $I, I_1, \ldots, I_k$ denote the sequence of databases produced by applying MDs according to Definition 3.2.1 starting from I such that I_k is stable. We say that (I, I_k) satisfy Σ and denote it as $(I, I_k) \models \Sigma$. A database may have many stable instances depending on the order of MD applications [10, 24].

Example 3.2.2. *Let (10,'Star Wars: Episode IV - 1977', 1977) and (40,'Star Wars: Episode III - 2005', 2005) be tuples in relation movies and ('Star Wars') be a tuple in relation highBudgetMovies whose schemas are shown in Table 1.1. Consider MD σ_1 : movies[title] $\approx$ highBudgetMovies[title] $\rightarrow$ movies[title] $\rightleftharpoons$ highBudgetMovies[title]. Let*

'Star Wars: Episode IV - 1977' $\approx$ *'Star Wars' and 'Star Wars: Episode III - 2005'* $\approx$ *'Star Wars' be true. Since the movies with titles 'Star Wars: Episode IV - 1977' and 'Star Wars: Episode III - 2005' are different movies with distinct titles, one can unify the title in the tuple ('Star Wars') in highBudgetMovies with only one of them in each stable instance. Each alternative leads to a distinct instance.*

MDs may *not* be precise. If two values are declared similar according to an MD, it does not mean that they represent the same real-world entities. However, it is more likely for them to represent the same value than those that do not match an MD. Since it may be cumbersome to develop complex MDs that are sufficiently accurate, researchers have proposed systems that automatically identify MDs from the database content [41].

3.3 Conditional Functional Dependencies

Users usually define integrity constraints (ICs) to ensure the quality of the data. Conditional functional dependencies (CDFs) have been useful in defining quality rules for cleaning data [22, 61, 30, 15, 61, 23]. They extend functional dependencies, which are arguably the most widely used ICs [27]. Relation R with sets of attributes X and Y satisfies FD $X \to Y$ if every pair of tuples in R that agree on the values of X will also agree on the values of Y. A CFD ϕ over R is a form $(X \to Y, t_p)$ where $X \to Y$ is an FD over R and t_p is a tuple pattern over $X \cup Y$. For each attribute $A \in X \cup Y$, $t_p[A]$ is either a constant in the domain of A or an unnamed variable denoted as '-' that takes values from the domain of A. The attributes in X and Y are separated by || in t_p.

For example, consider relation *mov2locale*(*title*, *language*, *country*) in BOM. The CFD ϕ_1: (*title*, *language* $\to$ *country*, (-, English || -)) indicates that *title* uniquely identifies *country* for tuples whose language is English. Let $\asymp$ be a predicate over data values and unnamed variable '-', where $a \asymp b$ if either $a = b$ or a is a value and b is '-'. The predicate $\asymp$ naturally extends to tuples, for example, ('Bait', English, USA) $\asymp$ ('Bait', -, USA). Tuple t_1 *matches* t_2 if $t_1 \asymp t_2$. Relation R satisfies the CFD $(X \to Y, t_p)$ iff for each pair of tuples t_1, t_2 in the instance if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$. In other words, if $t_1[X]$ and $t_2[X]$ are equal and match pattern $t_p[X]$, $t_1[Y]$ and $t_2[Y]$ are equal and match $t_p[Y]$.

A relation satisfies a set of CFDs Φ, if it satisfies every CFD in Φ. For each set of CFDs Φ, we can find an equivalent set of CFDs whose members have a single attribute

on their right-hand side [15, 61, 23]. For the remainder of this thesis, we assume that each CFD has a single attribute on its right-hand side.

CFDs may be violated in real-world and heterogeneous datasets [61, 30]. For example, the pair of tuples r_1 :('Bait', English, USA) and r_2 :('Bait', English, Ireland) in *movie2locale* violate ϕ_1. One can use attribute value modifications to repair violations of a CFD in a relation and generate a repaired relation that satisfies the CFD [12, 25, 60, 15, 61, 40, 23]. For instance, one may repair the violation of ϕ_1 in r_1 and r_2 by updating the value of title or language in one of the tuples to a value other than Bait or English, respectively.

One may also repair this violation by replacing the countries in these tuples with the same value. Inserting new tuples does not repair CFD violations and one may simulate tuple deletion using value modifications. Moreover, removing tuples leads to unnecessary loss of information for attributes that do not participate in the CFD. Modifying attribute values is sufficient to resolve CFD violations [15, 61]. Thus, given a pair of tuples t_1 and t_2 in R that violate CFD $(X \rightarrow A, t_p)$, to resolve the violation, one must either modify $t_1[A]$ (resp. $t_2[A]$) such that $t_1[A] = t_2[A]$ and $t_1[A] \asymp t_p[A]$, update $t_1[X]$ (resp. $t_2[X]$) such that $t_1[X]t_p[X]$ (resp. $t_2[X]t_p[X]$) or $t_1[X] \neq t_2[X]$. Let R be a relation that violates CFD ϕ. Each updated instance of R that is generated by applying the aforementioned repair operations and does not contain any violation of ϕ is a *repair* of R. As there are multiple fixes for each violation, there may be many repairs for each relation.

As opposed to FDs, a set of CFDs may be inconsistent; in other words, there is not any non-empty database that satisfies them [11, 15, 61, 23]. For example, the CFDs $(A \rightarrow B, a_1||b_1)$ and $(B \rightarrow A, b_1||a_2)$ over relation $R(A, B)$ cannot be both satisfied by any non-empty instance of R. The set of CFDs used in cleaning is consistent [11, 15, 61, 23]. We refer the reader to [11] for algorithms to detect inconsistent CFDs.

3.4 Semantic of Learning

3.4.1 Different Approaches

Let I be an instance of schema $\mathcal{S}$ with MDs Σ that violate some CFDs Φ. A *repair* of I is a stable instance of I that satisfies Φ. The values in I are repaired to satisfy Φ using the method explained in Section 3.3. Given I and a set of training examples E, we wish

to learn a definition for a target relation T in terms of the relations in $\mathcal{S}$. One may not learn an accurate definition by applying current learning algorithms over I as the algorithm may consider different occurrences of the same value to be distinct or learn patterns that are induced based on tuples that violate CFDs.

Let *StableInstances*(I, Σ) be the set of stable instances of I. One can learn definitions by generating all possible repairs of I, i.e., $\mathbf{J} = \textit{StableInstances}(I, \Sigma)$, learning a definition over each repair separately, and computing a union (disjunction) of all learned definitions. Since the discrepancies are resolved in repaired instances, this approach may learn accurate definitions.

However, this method is neither desirable nor feasible for large databases. As a large database may have numerous repairs, it takes a great deal of time and storage to compute and materialize all of them. Moreover, we have to run the learning algorithm once for each repair, which may take an extremely long time. Most importantly, as the learning has been done separately over each repair, it is *not* clear whether the final definition is sufficiently effective considering the information of all stable instances.

For example, let database I have two repairs I_1^s and I_2^s over which the aforementioned approach learns definitions H_1 and H_2, respectively. H_1 and H_2 must cover a relatively small number of negative examples over I_1^s and I_2^s, respectively. However, H_1 and H_2 may cover many negative examples over I_2^s and I_1^s, respectively. Thus, the disjunction of H_1 and H_2 will *not be effective* considering the information in both I_1^s and I_2^s. Hence, it is not clear whether the disjunction of H_1 and H_2 is the definition that covers all positive and the fewest negative examples over I_2^s and I_1^s. Moreover, it is not clear how to encode the final result as we may end up with numerous definitions, each of which is accurate over one stable instance.

Another approach is to consider only the information shared among all repairs for learning. The resulting definition will cover all positive and the fewest negative examples considering the information common among all repaired instances. This approach has been used in the context of query answering over inconsistent data (i.e., consistent query answering [6, 10]). However, this approach may lead to many positive and negative examples being ignored, as their connections to other relations in the database may *not* be present in all stable instances.

For example, consider the tuples in relations *movies* and *highBudgetMovies* in Example 3.2.2. The training example *('Star Wars')* has different values in different stable

instances of the database; therefore, it will be ignored. It will also be connected to two distinct movies with vastly different properties in each instance. Similarly, repairing the instance to satisfy the violated CFDs may further reduce the amount of training examples shared among all repairs. The training examples are usually costly to obtain, and the lack of sufficient training examples may result in inaccurate learned definitions. In a sufficiently heterogeneous database, most positive and negative examples may not be common among all repairs; thus, the learning algorithm may learn an inaccurate or simply an empty definition.

Therefore, we opt for a middle-ground. We follow the approach of learning directly over the original database. However, we also give the language of definitions and semantic of learning enough flexibility to take advantage of as much (training) information as possible. Each definition will be a compact representation of a set of definitions, each of which is sufficiently accurate over some repairs. If one increases the expressivity of the language, learning and checking coverage for each clause may become inefficient [20]. We ensure that the added capability to the language of definitions is minimal, so learning remains efficient. Next, we present our modifications to the hypothesis space and semantic of learning.

3.4.2 Heterogeneity in Definitions

We represent the heterogeneity of the underlying data in the language of the learned definitions. Each new definition encapsulates the definitions learned over the repairs of the underlying database. Thus, we add the similarity operation, $x \approx y$, to the language of Horn definitions. We also add a set of new (built-in) relation symbols V_c with arity two called *repair relations* to the set of relation symbols used by the Datalog definitions over schema $\mathcal{S}$. A literal with a repair relation symbol is a *repair literal.* Each repair literal $V_c(x, v_x)$ in a definition H represents replacing the variable (or constant) x in (other) existing literals in H with variable v_x if condition c holds. Condition c is a conjunction of $=$, $\neq$, and $\approx$ relations over the variables and constants in the clause. Each repair literal reflects a repair operation explained in Sections 3.2 and 3.3 for an MD or violated CFD over the underlying database. The condition c is computed according to the corresponding MD or CFD. Finally, we add a set of literals with $=$, $\neq$, and $\approx$ relations called *restriction literals* to establish the relationship between the re-

placement variables, e.g, v_x, according to the corresponding MDs and CFDs. Consider again the database created by integrating IMDb and BOM datasets, schema fragments which are represented in Table 1.1, with MD σ_1 : *movies[title]* $\approx$ *highBudgetMovies[title]* $\rightarrow$ *movies[title]* $\rightleftharpoons$ *highBudgetMovies[title]*. We may learn the following definition for the target relation *highGrossing*.

$$\begin{aligned} \textit{highGrossing}(x) \leftarrow & \textit{movies}(y,t,z), \textit{mov2genres}(y,\text{`}\textit{comedy}\text{'}), \\ & \textit{highBudgetMovies}(x), x \approx t,\, V_{x\approx t}(x,v_x), \\ & V_{x\approx t}(t,v_t), v_x = v_t. \end{aligned}$$

The repair literals $V_{x\approx t}(x,v_x)$ and $V_{x\approx t}(t,v_t)$ represent the repairs applied to x and t to produce a unified value according to σ_1. We add equality literal $v_x = v_t$ to restrict the replacements according to the corresponding MD.

We also use repair literals to fix a violation of a CFD in a clause. These repair literals reflect the operations explained in Section 3.3 to fix the violation of a CFD in a relation. The resulting clause represents possible repairs for a violation of a CFD in the clause. A variable may appear in multiple literals in the body of a clause and some repairs may modify only some of the occurrences of the variable (e.g., the example on the BOM database in Section 3.3). Thus, before adding repair literals for both MDs and CFDs, we replace each occurrence of a variable with a fresh one and add equality literals (i.e., *induced equality literals*), to maintain the connection between their replacements. Similarly, we replace each occurrence of the constant with a fresh variable and use equality literals to set the value of the variable equal to the constant in the clause.

Example 3.4.1. *Consider the following clause, which may be a part of a learned clause over the integrated IMDb and BOM database for highGrossing.*

$$\begin{aligned} \textit{highGrossing}(x) \leftarrow & \textit{mov2locale}(x, English, z), \\ & \textit{mov2locale}(x, English, t). \end{aligned}$$

This clause reflects a violation of CFD ϕ_1 from Section 3.3 in the underlying database as it indicates that English movies with the same title are produced in different countries. We first replace each occurrence of repeated variable x with a new variable and then add

the repair literals. Due to the limited space available, we do not show the repair literals and their conditions for modifying the values of constant 'English'. Let condition c be $x_1 = x_2 \wedge z \neq t$.

$$\begin{aligned} highGrossing(x_1) \leftarrow\ & mov2locale(x_1, English, z), \\ & mov2locale(x_2, English, t), x_1 = x_2,\ V_c(x_1, v_{x_1}), \\ & V_c(x_2, v_{x_2}), v_{x_1} \neq x_2, v_{x_2} \neq x_1,\ V_c(z, t), \\ & V_c(t, z),\ V_c(z, v_z),\ V_c(t, v_t), v_z = v_t. \end{aligned}$$

We call a clause (definition) *repaired* if it does *not* have any repair literal. Each clause with repair literals represents a set of repaired clauses. We convert a clause with repair literals into a set of repaired clauses by iteratively applying repair literals to and eliminating them from the clause. To apply a repair literal $V_c(x, v_x)$ to a clause, we first evaluate c considering the (restriction) literals in the clause. If c holds, we replace all occurrences of x with v_x in all literals and the conditions of the other repair literals in the clause and remove $V_c(x, v_x)$; otherwise, we only eliminate $V_c(x, v_x)$ from the clause. We progressively apply all repair literals until no repair literal is left. Finally, we remove all restriction and induced equality literals that contain at least one variable that does not appear in any literal with a schema relation symbol. The resulting set is called the *repaired clauses* of the input clause.

Example 3.4.2. *Consider the following clause over the movie database of IMDb and BOM.*

$$\begin{aligned} highGrossing(x) \leftarrow\ & movies(y, t, z), mov2genres(y, \text{`}comedy\text{'}), \\ & highBudgetMovies(x), x \approx t,\ V_{x \approx t}(x, v_x), \\ & V_{x \approx t}(t, v_t), v_x = v_t. \end{aligned}$$

The application of repair literals $V_{x \approx t}(x, v_x)$ *and* $V_{x \approx t}(t, v_t)$ *results in the following clause:*

$$\begin{aligned} highGrossing(v_x) \leftarrow\ & movies(y, v_t, z), mov2genres(y, \text{`}comedy\text{'}), \\ & highBudgetMovies(v_x), v_x = v_t. \end{aligned}$$

Similarly to the repair of a database based on MDs and CFDs, the application of a set of repair literals to a clause may create multiple repaired clauses depending on the order in which the repair literals are applied.

Example 3.4.3. *Consider a target relation* $T(A)$*, an input database with schema* $\{R(B), S(C)\}$*, and MDs* $\phi_1 : T[A] \approx R[B] \rightarrow T[A] \rightleftharpoons R[B]$ *and* $\phi_2 : T[A] \approx S[C] \rightarrow T[A] \rightleftharpoons S[C]$*. The definition* $H : T(x) \leftarrow R(y), x \approx y, V_{x \approx y}(x, v_x)$*,* $V_{x \approx y}(y, v_y), v_x = v_y, S(z), x \approx z$*,* $V_{x \approx z}(x, u_x)$*,* $V_{x \approx z}(z, v_z), u_x = v_z$*. over this schema has two repaired definitions:* $H'_1 : T(v_x) \leftarrow R(v_y), v_x = v_y, S(z)$*. and* $H'_2 : T(u_x) \leftarrow R(y), S(v_z), u_x = v_z$*. As another example, the application of each repair literal in the clause of Example 3.4.1 results in a distinct repaired clause. For instance, applying* $V_c(x_1, v_{x_1})$ *replaces* x_1 *with* v_{x_1} *in all literals and conditions of the repair literals and results in the following.*

$$\begin{aligned} \mathit{highGrossing}(v_{x_1}) \leftarrow\ & \mathit{mov2locale}(v_{x_1}, \mathit{English}, z), \\ & \mathit{mov2locale}(x_2, \mathit{English}, t),\ V_c(x_2, v_{x_2}), v_{x_1} \neq x_2. \end{aligned}$$

As Example 3.4.3 illustrates, repair literals provide a compact representation of multiple learned clauses where each may explain the patterns in the training data in some repair of the input database. Given an input definition H, the *repaired definitions* of H are a set of definitions where each definition contains exactly one repaired clause per each clause in H.

In this process, we have to keep both the similarity condition in generalization, so we retain the information about the underlying constants and their relationships and consequently the correct repairs. We also need to keep track of the substituting variables and their restrictions because we need to keep all possible repairs of the remaining literals after dropping other literals that have some repairs in common with the remaining ones during generalization.

3.4.3 Coverage over Heterogeneous Data

A learning algorithm evaluates the score of a definition according to the number of the positive and negative examples it covers. One way to measure the score of a definition is to compute the difference between the number of positive and negative examples covered by that definition [17, 54, 62]. Each definition may have multiple repaired definitions,

each of which may cover a different number of positive and negative examples of the repairs of the underlying database. Thus, it is not clear how the score of a definition should be computed.

One approach is to consider that a definition covers a positive example if at least one of its repaired definitions covers it in some repaired instances. Given that all other conditions are the same, this approach may lead to learning a definition with numerous repaired definitions where each may not sufficiently cover many positive examples. Hence, it is not clear whether each repaired definition is accurate. A more restrictive approach is to consider that a definition covers a positive example if all its repaired definitions cover it. This method will deliver a definition whose repaired definitions have high positive coverage over repaired instances. There are similar alternatives for defining coverage of negative examples. One may consider that a definition covers a negative example if all of its repaired definitions cover it. Thus, if at least one repaired definition does not cover the negative example, the definition will not cover it. This approach may lead to learning numerous repaired definitions that cover many negative examples. In contrast, a restrictive approach may define a negative example covered by a definition if at least one of its repaired definitions covers it. In this case, each learned repaired definition will generally *not* cover an excessive number of negative examples. We adopt a more restrictive approach.

Definition 3.4.4. *A definition H covers a positive example e with regard to database I iff every repaired definition of H covers e in some repairs of I.*

Definition 3.4.5. *A definition H covers a negative example e with regard to database I if at least one of the repaired definitions of H covers e in some repairs of I.*

Example 3.4.6. *Consider again the schema, MDs, and definition H in Examples 3.4.3 and the database of this schema with training example $T(a)$ and tuples $\{R(b), S(c)\}$. Assume that $a \approx b$ and $a \approx c$ are true. The database has two stable instances I'_1 : $\{T(v_{a,b}), R(v_{a,b}),\ S(c)\}$ and I'_2 : $\{T(v_{a,c}), R(b),\ S(v_{a,c})\}$. Definition H covers the single training example in the original database according to Definition 3.4.4 as its repaired definitions H'_1 and H'_2 cover the training example in repaired instances I'_1 and I'_2, respectively.*

Definition 3.4.4 provides a more flexible semantic than considering only the common information between all repaired instances, as described in Section 3.4.1. The latter semantic considers that the definition H covers a positive example if it covers that example in all repaired instances of a database.

As explained in Section 3.4.1, our semantic of coverage hits a middle-ground between the approaches of using only the information common between all repaired instances and learning over each repaired instance separately and returning the union of the results. It avoids ignoring too many positive examples and ensures that the learned definition does *not* cover too many negative examples overall.

Chapter 4: Methods

4.1 DLearn

This section proposes a learning algorithm called *DLearn* for efficiently learning over heterogeneous data. DLearn follows the approach used in the bottom-up relational learning algorithms [50, 51, 48, 53]. In this approach, the *LearnClause* function in Algorithm 1 has two steps: It first builds the most specific clause in the hypothesis space that covers a given positive example, called a *bottom-clause*. Then, it generalizes the bottom-clause to cover as many positive and as fewest negative examples as possible. DLearn extends these algorithms by integrating the input MDs and CFDs into the learning process to learn over heterogeneous data.

4.1.1 Bottom-Clause Construction

A *bottom-clause* C_e associated with an example e is the most specific clause in the hypothesis space that covers e relative to the underlying database I. Let I be the input database of schema $\mathcal{S}$ and the set of MDs Σ and CFDs Φ. The bottom-clause construction algorithm consists of two phases. First, it finds all the information in I relevant to e. The information relevant to example e is the set of tuples $I_e \subseteq I$ that are connected to e. A tuple t is connected to e if we can reach t using a sequence of exact or approximate (similarity) matching operations, starting from e. Given the information relevant to e, DLearn creates the bottom-clause C_e.

Example 4.1.1. *Given example highGrossing(Superbad), database in Table 4.1, and MD* σ_2 : $highGrossing[title] \approx movies[title] \rightarrow highGrossing[title] \rightleftharpoons movies[title]$, *DLearn finds the relevant tuples movies(m1, Superbad (2007), 2007), mov2genres(m1, comedy), mov2countries(m1, c1), englishMovies(m1),*
mov2releasedate(m1, August, 2007), and countries(c1, USA). As the movie title in the training example, for example, Superbad, does not exactly match with the movie title in the movie's relation, for example, Superbad (2007), the tuple movies(m1, Superbad

movies(m1,Superbad (2007),2007)	mov2genres(m1,comedy)
movies(m2,Zoolander (2001),2001)	mov2genres(m2,comedy)
movies(m3,Orphanage (2007),2007)	mov2genres(m3,drama)
mov2countries(m1,c1)	countries(c1,USA)
mov2countries(m2,c1)	countries(c2,Spain)
mov2countries(m3,c2)	englishMovies(m1)
mov2releasedate(m1,August,2007)	englishMovies(m2)
mov2releasedate(m2,September,2001)	senglishMovies(m3)

Table 4.1: Example movie database

(2007), 2007) is obtained through an approximate match and similarity search according to σ_2. We obtain others via exact matches.

To find the information relevant to e, DLearn uses Algorithm 2. It maintains a set M that contains all seen constants. Let $e = T(a_1, \ldots, a_n)$ be a training example. First, DLearn adds $a_1, \ldots, a_n$ to M. These constants are values that appear in tuples in I. Then, DLearn searches all tuples in I that contain at least one constant in M and adds them to I_e. For an exact search, DLearn uses simple SQL selection queries over the underlying relational database. For similarity search, DLearn uses MDs in Σ. If M contains constants in some relation R_i and given an MD $\sigma' \in \Sigma$, $\sigma' : R_1[A_{1\ldots n}] \approx R_2[B_{1\ldots n}] \rightarrow R_1[C] \rightleftharpoons R_2[D]$ DLearn performs a similarity search over $R_2[B_j]$, $1 \leq j \leq n$ to find relevant tuples in R_2, denoted by $\psi_{B_i \approx M}(R_2)$. We store the pairs of tuples that satisfy the similarity match in I_e in a table in main memory. We discuss the details of the implementation of DLearn over relational database systems in Section 4.2. For each new tuple in I_e, the algorithm extracts new constants and adds them to M. It repeats this process for a fixed number of iterations d.

To create the bottom-clause C_e from I_e, DLearn first maps each constant in M to a new variable. It creates the head of the clause by creating a literal for e and replacing the constants in e with their assigned variables. Then, for each tuple $t \in I_e$, DLearn creates a literal and adds it to the body of the clause, replacing each constant in t with its assigned variable. If there is a variable that appears in more than a single literal, we add the equality literals according to the method explained in Section 3.4.2. If t satisfies a similarity match according to the table of similarity matches with tuple t', we add a similarity literal s per each value match in t and t' to the clause.

Let σ be the corresponding MD of this similarity match. We will also add repair

Algorithm 2: DLearn bottom-clause construction algorithm.

Input : example e, # of iterations d
Output: bottom-clause C_e

```
1  I_e = {}
2  M = {} // M stores known constants
3  add constants in e to M
4  for i = 1 to d do
5  |  foreach relation R ∈ I do
6  |  |  foreach attribute A in R do
7  |  |  |  // select tuples with constants in M
8  |  |  |  I_R = σ_{A∈M}(R)
9  |  |  |  if ∃ MD σ' ∈ Σ, σ' : R_1[A_{1...n}] ≈ R_2[B_{1...n}] → R_1[C] ⇌ R_2[D] then
10 |  |  |  |  I_R = I_R ∪ ψ_{B_j≈M}(R), 1 ≤ j ≤ n
11 |  |  |  foreach tuple t ∈ I_R do
12 |  |  |  |  add t to I_e and constants in t to M
13 C_e = create clause from e and I_e
14 return C_e
```

literals $V_s(x, v_x)$ and $V_s(y, v_y)$ and restriction equality literal $v_x = v_y$ to the clause according to *sigma*.

Example 4.1.2. *Given the relevant tuples found in Example 4.1.1, DLearn creates the following bottom-clause:*

$$\begin{aligned} &highGrossing(x) \leftarrow movies(y, t, z), x \approx t,\ V_{x \approx t}(x, v_x),\\ &\quad V_{x \approx t}(t, v_t), v_x = v_t, mov2genres(y, \text{`}comedy\text{'}),\\ &\quad mov2countries(y, u), countries(u, \text{`}USA\text{'}),\\ &\quad englishMovies(y), mov2releasedate(y, \text{`}August\text{'}, w). \end{aligned}$$

Then, we scan C_e to find violations of each CFD in Φ and add their corresponding repair literals. Since each CFD is defined over a single table, we first group literals in C_e based on their relation symbols. For each group with the relation symbol R and CFD ϕ on R, our algorithm scans the literals in the group, finds every pair of literals that violate ϕ, and adds the repair and restriction literals to the group.

We add the repair and restriction literals corresponding to the repair operations

explained in Section 3.3 to the group and consequently C_e, as illustrated in Example 3.4.1.

The added repair literals will not induce any new violation of ϕ in the clause [15, 61, 23]. However, repairing a violation of ϕ may induce violations for anther CFD ϕ' over R [23]. For example, consider CFD $\phi_3 : (A \rightarrow B, - \mid\mid -)$ and $\phi_4 : (B \rightarrow C, - \mid\mid -)$ on relation $R(A, B, C)$. Given literals $l_1 : R(x_1, y_1, z_1)$ and $l_2 : R(x_1, y_1, z_2)$ that violate ϕ_4, our method adds repair literals that replaces y_1 in l_1 with a fresh variable. This repair literal produces a repaired clause that violates ϕ_3. Thus, the algorithm repeatedly scans the clause and adds repair and restriction literals to it for all CFDs until there is a repair for every violation of CFDs, both those in the original clause and those induced by the repair literals added in the preceding iterations. The repaired literals for the violations induced by other repair literals will use the replacement variables from the violating repair literals as their arguments and conditions.

It may take a long time to generate the clause that contains all repair literals for all original and induced violations of every CFD in a large input bottom-clause. Hence, we reduce the number of repair literals per CFD violation by adding only the repair literals for the variables of the right-hand side attribute of the CFD that use current variables in the violation. For instance, in Example 3.4.1, the algorithm does *not* introduce literals $V_c(z, v_z)$, $V_c(t, v_t)$, and $v_z = v_t$ and only uses literals $V_c(z, t)$ and $V_c(t, z)$ to repair the clause in Example 3.4.1. The repair literals for the variables corresponding to the left-hand side of the CFD will be used as explained previously. This approach follows the popular minimal repair semantic for repairing CFDs [12, 25, 60, 15, 61, 40, 23], as it repairs the violation by modifying fewer variables than the repair literals that introduce fresh variables to the both literals of the violation (e.g., the one versus two modifications induced by $V_c(z, v_z)$, $V_c(t, v_t)$ in the repair of the clause in Example 3.4.1). Since each CFD is defined over a single relation, the aforementioned steps are applied separately to literals of each relation, which are usually a considerably smaller set than the set of all literals in the bottom-clause. Moreover, the bottom-clause is significantly smaller than the size of the entire database. Thus, the bottom-clause construction algorithm takes significantly less time than producing the repairs of the underlying database.

Current bottom-clause constructions methods do not induce inequality *neq* literal between distinct constants in the database and their corresponding variables and represent their relationship by replacing them with distinct variables. If the inequality literal is used, the eventual generalization of the bottom-clause may be too strict and lead to a

learned clause that does not cover a sufficient number of positive examples [51, 49, 17, 53]. For example, let $T(x) :- R(x,y), S(x,z), y \neq z$. be a bottom-clause. This clause will not cover positive examples such as $T(a)$ for which we have $T(a) :- R(a,b), S(a,b)$. However, the bottom-clause $T(x) :- R(x,y), S(x,z)$ has more generalization power and may cover both positive examples such as $T(a)$ and $T(c)$ such that $T(c) :- R(c,b), S(c,d)$. As the goal of our algorithm aims to simulate relational learning over repaired instances of the original database, we follow the same approach and remove the inequality literals between variables. As our repair operations ensure that the arguments of inequality literals are distinct variables, our method exactly emulates bottom-clause construction in relational learning. The inequalities remain in the condition c of each repair literal V_c and will return true if the variables are distinct and there is no equality literal between them in the body of the clause and false otherwise. They are not used in learning and are used to apply repair literals on the final clause.

Proposition 4.1.3. *The bottom-clause construction algorithm for positive example e and database I with MDs Σ and CFD Φ terminates. Moreover, the bottom-clause C_e created from I_e using the algorithm covers e.*

4.1.2 Generalization

After creating the bottom-clause C_e for example e, DLearn generalizes C_e to produce a clause that is *more general* than C_e. Clause C is more general than clause D if and only if C covers at least all positive examples covered by D. A more general clause than C_e may cover more positive examples than C_e. DLearn iteratively applies the generalization to find a clause that covers the most positive and fewest negative examples as possible. It extends the algorithm in ProGolem [51] to produce generalizations of C_e in each step efficiently. This algorithm is based on the concept of θ-subsumption, which is widely used in relational learning [17, 49, 51]. We first review the concept of θ-subsumption for repaired clauses [17, 51], then, we explain how to extend this concept and its generalization methods for non-stable clauses.

Repaired clause C *θ-subsumes* repaired clause D, denoted by $C \subseteq_\theta D$, iff there is some substitution θ such that $C\theta \subseteq D$ [17, 2]; in other words, the result of applying substitution θ to literals in C creates a set of literals that is a subset of or equal to the set of literals in D. For example, clause C_1 : $highGrossing(x) \leftarrow movies(x, y, z)$ θ-

subsumes $C_2 : highGrossing(a) \leftarrow movies(a, b, c), mov2genres(b, \text{'}comedy\text{'})$ as for substitution $\theta = \{x/a,\ y/b,\ z/c\}$, we have $C_1\theta \subseteq C_2$. We call each literal L_D in D where there is a literal L_C in C such that $L_C\theta = L_D$ a *mapped* literal under θ. For Horn definitions, we have C θ-subsumes D iff $C \models G$; in other words, C logically entails D [2, 17]. Thus, θ-subsumption is sound for generalization. If clauses C and D contain equality and similarity literals, the subsumption checking requires additional testing, which can be done efficiently [2, 17, 4]. Roughly speaking, current learning algorithms generalize a clause D efficiently by eliminating some of its literals which produces a clause that θ-subsumes D. We define θ-subsumption for clauses with repair literals using its definition for the repaired ones. Given a clause D, a repair literal $V_c(x, v_x)$ in D is *connected to* a non-repair literal L in D iff x or v_x appear in L or in the arguments of a repair literal connected to L.

Definition 4.1.4. *Let $V(C)$ denote the set of all repair literals in C θ-subsumes D, denoted by $C \subseteq_\theta D$, iff*

- *there is some substitution θ such that $C\theta \subseteq D$ where repair literals are treated as normal ones and*
- *every repair literal connected to a mapped literal in D is also a mapped literal under θ.*

Definition 4.1.4 ensures that each repair literal that modifies a mapped one in D has a corresponding repair literal in C. Intuitively, this ensures that there is subsumption mapping between corresponding repaired versions of C and D. The next step is to examine whether θ-subsumption provides a sound bases for generalization of clauses with repair literals. We first define logical entailment following the semantics of Definition 3.4.4.

Definition 4.1.5. *We have $C \models D$ if and only if there is an onto relation f from the set of repairs of C to the one of D such that for each repaired clause of C, C_r, and each $D_r \in f(C_r)$, we have $C_r \models D_r$.*

According to Definition 4.1.5, if one wants to follow the generalization method used in the current learning algorithm to check whether C generalizes D, one must enumerate and check the θ-subsumption of almost every pair of repaired clauses of C and D in the worst case. Since both clauses normally contain many literals and θ-subsumption

is NP-hard [2], this method is not efficient. The problem is more complex if one wants to generalize a given clause D. It may have to generate all repaired clauses of D and generalize each of them separately. It is not clear how all produced repaired clauses can be unified and represented in a single non-repaired one. The hypothesis space quickly explodes if we cannot represent them in a single clause, as the algorithm may have to keep track of and generalize almost as many clauses as repairs of the underlying database. Moreover, because the learning algorithm performs numerous generalizations and coverage tests, learning a definition may take a long time. The following theorem establishes that θ-subsumption is sound for generalization of clauses with repair literals.

Theorem 4.1.6. *Given clauses C and D, if C θ-subsumes D, we have $C \models D$.*

To generalize C_e, DLearn randomly selects a subset $E^{+s} \subseteq E^{+}$ of positive examples. For each example e' in E^{+s}, DLearn generalizes C_e to produce a candidate clause C', which is more general than C_e and covers e'. Given clause C_e and positive example $e' \in E^{+s}$, DLearn produces a clause that θ-subsumes C_e and covers e' by removing the *blocking literals*. It first creates a total order between the relation symbols and the symbols of repair literals in the schema of the underlying database, for example, by using a lexicographical order and adding the condition and argument variables to the symbol of the repair literals. Thus, it establishes an order in each clause in the hypothesis space. Let $C_e = T \leftarrow L_1, \cdots, L_n$ be the bottom-clause. The literal with relation symbol L_i is a *blocking literal* if and only if i is the least value such that for all substitutions θ where $e' = T\theta$, $(T \leftarrow L_1, \cdots, L_i)\theta$ does not cover e' [51].

Example 4.1.7. *Consider the bottom-clause C_e in Example 4.1.2 and positive example e' = highGrossing('Zoolander'). To generalize C_e to cover e', DLearn drops the literal mov2releasedates(y, 'August', u) because the movie Zoolander was not released in August.*

DLearn removes all blocking literals in C_e to produce the generalized clause C'. DLearn also ensures that all literals in the resulting clause are head-connected. For example, if a non-repair literal L is dropped, so are the repair literals whose only connection to the head literal is through L. Since C' is generated by dropping literals, it θ-subsumes C_e. It also covers e' by construction. DLearn generates one clause per example in E^{+s}. From the set of generalized clauses, DLearn selects the highest scoring candidate clause. The score of a clause is the number of positive minus the number of negative examples

covered by the clause. DLearn then repeats this with the selected clause until its score is not improved.

During each generalization step, the algorithm should ensure that the generalization is minimal with respect to θ-subsumption; in other words, there is *not* any other clause G such that G θ-subsumes C_e and C' θ-subsumes G [51]. Otherwise, the algorithm may miss some effective clauses and produce a clause that is overly general and may cover too many negative examples. The following proposition states that DLearn produces a minimal generalization in each step:

Proposition 4.1.8. *Let C be a head-connected and ordered clause generated from a bottom-clause using DLearn generalization algorithm. Let clause D be the generalization of C produced in a single generalization step by the algorithm. Given the clause F that θ-subsumes C, if D θ-subsumes F, then D and F are equivalent.*

4.1.3 Efficient Coverage Testing

DLearn checks whether a candidate clause covers training examples to determine blocking literals in a clause. It also computes the score of a clause by computing the number of training examples covered by the clause. Coverage tests dominate the time for learning [17]. One approach to performing a coverage test is to transform the clause into an SQL query and evaluate it over the input database to determine the training examples covered by the clause. However, since bottom-clauses over large databases normally have many literals (e.g., some may have hundreds), the SQL query will involve long joins, making the evaluation extremely slow. Furthermore, it is challenging to evaluate clauses using this approach over heterogeneous data [10]. It is also not clear how to evaluate clauses with repair literals.

We use the concept of θ-subsumption for clauses with repair literals and the result of Theorem 4.1.6 to compute coverage efficiently. To evaluate whether C covers a positive example e over database I, we first build a bottom-clause G_e for e in I called a *ground bottom-clause*. Then, we check whether $C \wedge I \models e$ using θ-subsumption. We first check whether $C \subseteq_\theta G_e$. Based on Theorem 4.1.6, if we find a substitution θ for C such that $C\theta \subseteq G_e$, and C logically entails G_e, C thus covers e. However, if we cannot find such a substitution, it is not clear whether C logically entails G_e as Theorem 4.1.6 does not provide the necessity of θ-subsumption for logical entailment. Fortunately, this is true if

we have only repair literals for MDs in C and G_e.

Theorem 4.1.9. *Given clauses C and D such that every repair literal in C and D corresponds to an MD, if $C \models D$, C θ-subsumes D.*

We leverage Theorem 4.1.9 to check whether C covers e efficiently as follows: Let C^{md} and G_e^{md} be the clauses that have the same head literal as C and G_e and contain all body literals in C and G_e without any connected repair literal and those where all their connected repair literals correspond to some MDs, respectively. Thus, if there is no subsumption between C and G_e, our algorithm attempts to find a subsumption between C^m and G_e^m. If there is no subsumption mapping between C^m and G_e^m, C does not cover e. Otherwise, let C^{cfd} and G_e^{cfd} be the set of body literals of C and G_e that do not appear in the body of C^{md} and G_e^{md}, respectively. We apply the repair literals in C^{cfd} and G_e^{cfd} in C and D and perform subsumption checking for pairs of resulting clauses. If every obtained clause of C θ-subsumes at least one resulting clause of G_e, C covers e; otherwise, C does not cover e. We note than the resulting clauses are not repairs of C and G_e, as they still have the repair literals that correspond to some MD.

We follow a similar method to the one explained in the preceding paragraph to check whether clause C covers a negative example, except we use the semantic introduced in Definition 3.4.5 to determine the coverage of negative examples. Let G_{e^-} be the ground bottom-clause for the negative example e^-. We generate all repaired clauses of the clause C as described in Section 3.4. Then, we check whether each repaired clause of C θ-subsumes G_{e^-} the same way as checking θ-subsumption for C and a ground bottom-clause for a positive example. C θ-subsumes G_{e^-} as soon as one repaired clause of C θ-subsumes G_{e^-}.

Proposition 4.1.10. *Given the clause C and ground bottom-clause G_{e^-} for negative example e^- relative to database I, clause C covers e^- iff a repair of C θ-subsumes G_e^-.*

4.1.4 Commutativity of Cleaning and Learning

An interesting question is whether our algorithm produces the same answer as the one that learns a repaired definition over each repair of I separately. We show that, our algorithm delivers similar information as the one that separately learns over each repaired instance. Thus, our algorithm learns using the compact representation without any loss

of information. Let $\textit{RepairedCls}(C)$ denote the set of all repaired clauses of clause C. Let $BC(e,\ I, \Sigma,\ \Phi)$ denote the bottom-clause generated by applying the bottom-clause construction algorithm in Section 4.1.1 using example e over database I with the set of MDs Σ and CFDs Φ. Also, let $BC_r(e,\ \textit{RepairedInst}(I, \Sigma, \Phi))$ be the set of repaired clauses generated by applying the bottom-clause construction to each repair of I for e.

Theorem 4.1.11. *Given database I with MDs Σ, CFDs Φ and set of positive examples E^+, $\forall e \in E^+$ $BC_r(e, \textit{RepairedInst}(I, \Sigma, \Phi)) = \textit{RepairedCls}(BC(e, I, \Sigma))$.*

Thus, our algorithm considers exactly the same set of bottom-clauses as the one that learns over each repaired instance. We further prove that this set remains intact during generalization. Now, assume that $\textit{Generalize}(C, e',\ I, \Sigma,\ \Phi)$ denotes the clause produced by generalizing C to cover example e' over database I with the set of MDs Σ and CFDs Φ in a single step of applying the algorithm in Section 4.1.2. Give a set of repaired clauses $\mathbf{C}$, let $\textit{Generalize}_r\ (\mathbf{C}, e', \textit{RepairedInst}\ (I, \Sigma, \Phi))$ be the set of repaired clauses produced by generalizing every repaired clause in $\mathbf{C}$ to cover example e' in some repair of I using the algorithm in Section 4.1.2.

Theorem 4.1.12. *Given database I with MDs Σ and set of positive examples E^+ $\textit{Generalize}_r(\textit{StableCls}(C), e', \textit{RepairedInst}(I,\ \Sigma, \Phi)) = \textit{RepairedCls}(\textit{Generalize}(C, I, e', \Sigma, \Phi))$.*

4.2 Implementation

DLearn is implemented on top of VoltDB, *voltdb.com*, a main-memory relational database management system. We use the indexing and query processing mechanisms of the database system to create the (ground) bottom-clauses efficiently. The set of tuples I_e that DLearn gathers to build a bottom-clause may be large if many tuples in I are relevant to e, particularly when learning over a large database. To overcome this problem, DLearn randomly samples from the tuples in I_e to obtain a smaller tuple set $I_e^s \subseteq I_e$ and creates the bottom-clause based on the sampled data [51, 53].. To do so, DLearn restricts the number of literals added to the bottom-clause per relation through a parameter called *sample size*. To implement similarity over strings, DLearn uses the operator defined as the average of the *Smith-Waterman-Gotoh* and the *Length* similarity functions. The *Smith-Waterman-Gotoh* function [31] measures the similarity

of two strings based on their local sequence alignments. The *Length* function computes the similarity of the length of two strings by dividing the length of the smaller string by the length of the larger string. To improve efficiency, we precompute the pairs of similar values.

Chapter 5: Experiments

We empirically investigate the following questions:

- Can DLearn learn over heterogeneous databases effectively and efficiently? (Section 5.1.3)
- What is the benefit of using MDs and CFDs during learning? (Section 5.1.3)
- How does the number of training examples affect DLearn's effectiveness and efficiency? (Section 5.2.4)
- How does sampling affect DLearn's effectiveness and efficiency? (Section 5.2.5)

5.1 Experimental Settings

5.1.1 Datasets

Datasets: We use the databases shown in Table 5.1.
IMDb + OMDb: The Internet Movie Database (IMDb) and Open Movie Database (OMDb) contain information about movies, such as their titles, year and country of production, genre, directors, and actors [16]. We learn the target relation *dramaRestricted-Movies(imdbId)*, which contains the *imdbId* of movies that are of the *drama* genre and

Name	#R	#T	#P	#N
IMDb	9	3.3M	100	200
OMDb	15	4.8M		
Walmart	8	19K	77	154
Amazon	13	216K		
DBLP	4	15K	500	1000
Google Scholar	4	328K		

Table 5.1: Numbers of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset.

are rated R. The *imdbId* is only contained in the IMDb database, the genre information is contained in both databases, and the rating information is only contained in the OMDb database. Moreover, we can specify an MD that matches movie titles on IMDb with movie titles in OMDb as follows:

$$IMDb.movies[title] \approx OMDb.movies[title] \rightarrow \\ IMDb.movies[title] \rightleftharpoons OMDb.movies[title].$$

We refer to this dataset with one MD as **IMDb + OMDb (one MD)**. We also create MDs that match cast members and writer names between the two databases. We refer to the dataset that contains the three MDs as **IMDb + OMDb (three MDs)**.
Walmart + Amazon: The Walmart and Amazon databases contain information about products, such as their brand, price, categories, dimensions, and weight [16]. We learn the target relation *upcOfComputersAccessories(upc)*, which contains the *upc* of products that fall into the *Computers Accessories* category. The *upc* is contained in the Walmart database and the information about categories of products is contained in the Amazon database. Therefore, in order to learn a definition for this relation, one needs information from both databases. We use an MD that connects the product names across the datasets.
DBLP + Google Scholar: The DBLP and Google Scholar databases contain information about academic papers, such as their titles, authors, and venue and year of publication [16]. The information in the Google Scholar database is not clean, complete, or consistent; for example, many tuples are missing the year of publication. Therefore, we aim to augment the information in the Google Scholar database with information from the DBLP database. We learn the target relation *gsPaperYear(gsId, year)*, which contains the Google Scholar id *gsId* and the *year* of publication of the paper as indicated in the DBLP database. We use two MDs that match titles and venues in datasets.

5.1.2 CFDs

We find four, six, and two CFDs for IMDb+OMDb, Amazon+Walmart, and DBLP+Google Scholar, respectively; for example, *id* determines *title* in Google Scholar. To evaluate the performance of DLearn on data that contains CFD violations, we randomly inject each

aforementioned dataset with varying proportions of CFD violations, p. For example, p of 5% means that 5% of the tuples in each relation violate at least one CFD.

5.1.3 Systems, Metrics, and Environment

We compare DLearn against three baseline methods to evaluate the handling of MDs over datasets with only MDs. These methods use *Castor*, a state-of-the-art relational learning system [53]. Castor is shown to scale to large databases and be generally more effective and robust than other relational learning systems over the relational data model. However, Castor does *not* learn over databases with MDs.

Castor-NoMD: We use Castor to learn over the original databases. It does not use any information from MDs.

Castor-Exact: We use Castor but allow the attributes that appear in an MD to be joined by exact joins. Therefore, this system uses information from MDs but only considers exact matches between values.

Castor-Clean: We resolve the heterogeneities between entity names in attributes that appear in an MD by matching each entity in one database with the most similar entity in the other database. We use the same similarity function used by DLearn. Once the entities are resolved, we use Castor to learn over the unified and clean database.

To evaluate the effectiveness and efficiency of the version of DLearn that supports both MDs and CFDs, **DLearn-CFD**, we compare it with a version of DLearn that supports only MDs and is run over a version of the database whose CFD violations are repaired, **DLearn-Repaired**. We obtain this repair using the minimal repair method, which is a popular approach for repairing CFDs [23]. This enables us to evaluate our method for each type of inconsistency separately.

We use F1 score to measure effectiveness of definitions, which is the harmonic average of the precision and recall. We perform 5-fold cross validation over all datasets and report the average F1 score and time over the cross validation. DLearn uses the parameter *sample size* to restrict the size of (ground) bottom-clauses. We fix *sample size* to 10. In Section 5.2.5, we evaluate the impact of this parameter on DLearn's effectiveness and efficiency. All systems use 16 threads to parallelize coverage testing. We use a server with 30 2.3GHz Intel Xeon E5-2670 processors, running CentOS Linux with 500GB of main memory.

Dataset	Metric	Castor-NoMD	Castor-Exact	Castor-Clean	DLearn		
					$k_m = 2$	$k_m = 5$	$k_m = 10$
IMDb + OMDb (one MD)	F1 score	0.47	0.59	0.86	0.90	**0.92**	**0.92**
	Time (m)	0.12	0.13	0.18	0.26	0.42	0.87
IMDb + OMDb (three MDs)	F1 score	0.47	0.82	0.86	0.90	**0.93**	0.89
	Time (m)	0.12	0.48	0.21	0.30	25.87	285.39
Walmart + Amazon	F1 score	0.39	0.39	0.61	0.61	0.63	**0.71**
	Time (m)	0.09	0.13	0.13	0.13	0.13	0.17
DBLP + Google Scholar	F1 score	0	0.54	0.61	0.67	0.71	**0.82**
	Time (m)	2.5	2.5	3.1	2.7	2.7	2.7

Table 5.2: Results of learning over all datasets with MDs. Number of top similar matches denoted by k_m.

5.2 Empirical Results

5.2.1 Handling MDs

Table 5.2 presents the results over all datasets using DLearn and the baseline systems. DLearn obtains a better F1 score than the baselines for all datasets. DLearn uses information in the input MDs to find relevant information from all databases. Castor-NoMD does not learn any definition in the DBLP + Google Scholar dataset. Over this dataset, Castor cannot access information from the DBLP database. As a result, it is not able to find a reasonable definition. Castor-Exact is able to learn a definition over all datasets. However, as it relies only on exact matches, the learned definitions are not as effective as the ones of DLearn. Castor-Clean outperforms the other baselines as integrating the input databases by using a simple entity resolution technique provides benefits for learning effective definitions. DLearn outperforms Castor-Clean in all datasets.

DLearn also learns effective definitions over heterogeneous databases efficiently. Using MDs enables DLearn to consider more patterns and to thus learn a more effective definition. For example, Castor-Clean learns the following definition regarding Walmart + Amazon:

$$upcComputersAccessories(v0) \leftarrow walmart_ids(v1, v2, v0),$$
$$walmart_title(v1, v9), v9 = v10,$$
$$walmart_groupname(v1, \text{“}Electronics - General\text{”}),$$
$$amazon_title(v11, v10), amazon_listprice(v11, v16).$$
(positove covered=29, negative covered=11)

$$upcComputersAccessories(v0) \leftarrow walmart_ids(v1, v2, v0),$$
$$walmart_title(v1, v6), v6 = v7, amazon_title(v8, v7),$$
$$amazon_category(v8, \text{“}ComputersAccessories\text{”}).$$
(positove covered=38, negative covered=4)

The definitions learned by DLearn with the same data are as follows:

$$upcComputersAccessories(v0) \leftarrow walmart_ids(v1, v2, v0),$$
$$walmart_title(v1, v9), v9 \approx v10,$$
$$amazon_title(v11, v10), amazon_itemweight(v11, v16),$$
$$amazon_category(v11, \text{“}ComputersAccessories\text{”}).$$
(positove covered=35, negative covered=5)

$$upcComputersAccessories(v0) \leftarrow walmart_ids(v1, v2, v0),$$
$$walmart_brand(v1, \text{“}Tribeca\text{”}).$$
(positove covered=8, negative covered=0)

The definition learned by DLearn has higher precision; they have a similar recall. Castor-Clean first learns a clause that covers many positive examples but is not the desired clause. This affects its precision. DLearn first learns the desired clause and then learns a clause that has high precision.

The effectiveness of the definitions learned by DLearn depends on the number of matches considered in MDs, denoted by k_m. In the Walmart + Amazon, IMDb + BOM (one MD), and DBLP + Google Scholar datasets, using a higher k_m value results in learning a definition with higher F1 score. Even though the number of incorrect matches by the similarity function may increase, DLearn is able to ignore these false matches during learning. When using multiple MDs or when learning a difficult concept, a high k_m value affects DLearn's effectiveness. In these cases, incorrect matches represent noise that affects DLearn's ability to learn an effective definition. DLearn's effectiveness is

slightly lower with higher values of k_m in the IMDb + OMDb (three MDs) dataset. Nevertheless, it still delivers a more effective definition that other methods. As the value of k_m increases so does the learning time. This is because DLearn must process more information.

One should find the right trade-off between the improvement in the quality of the learned definition and the learning time, for instance, by experimenting with different values. Our empirical study indicates that DLearn delivers effective definitions quickly with a sufficiently small k_m. Thus, one may start with relatively small values of k_m and increase its value if ones has enough time to learn a relatively more accurate definition.

Next, we evaluate the effect of sampling on DLearn's effectiveness and efficiency. We use the IMDb + OMDb (three MDs) dataset and fix $k_m = 2$ and $k_m = 5$. We use 800 positive and 1,600 negative examples for training, and 200 positive and 400 negative examples for testing. Figure 5.1 (middle and right) shows the F1 score and learning time of DLearn with $k_m = 2$ and $k_m = 5$, respectively, when varying the sample size. For both values of k_m, the F1 score does *not* change significantly with different sampling sizes. With $k_m = 2$, the learning time remains almost the same with different sampling sizes. However, with $k_m = 5$, the learning time increases significantly. Therefore, using a small sample size is sufficient to efficiently learn an effective definition.

5.2.2 Handling MDs and CFDs

Table 5.3 compares DLearn-Repaired and DLearn-CFD. Over all three datasets DLearn-CFD performs (almost) equal to or substantially better than the baseline at all levels of violation injection. Since DLearn-CFD learns over all possible repairs of violating tuples, it has more available information, and, consequently, its hypothesis space is a super-set of the one used by DLearn-Repaired. In most datasets, the difference is more significant as the proportion of violations increase. Both methods deliver less effective results when there are more CFD violations in the data. However, DLearn-CFD is still able to deliver reasonably effective definitions. We use $k_m = 10$ for DBLP+Google Scholar and Amazon+Walmart and $k_m = 5$ for IMDb+OMDb as it takes a long time to use $k_m = 5$ for the latter.

Dataset	Metric	DLearn-CFD			DLearn-Repaired		
		$p = 0.05$	$p = 0.10$	$p = 0.20$	$p = 0.05$	$p = 0.10$	$p = 0.20$
IMDb + OMDb (three MDs)	F1 score	**0.79**	**0.78**	**0.73**	0.76	0.73	0.50
	Time (m)	11.15	16.26	26.95	5.70	12.54	22.28
Walmart + Amazon	F1 score	**0.64**	**0.61**	0.54	0.49	0.52	**0.56**
	Time (m)	0.17	0.2	0.23	0.18	0.18	0.19
DBLP + Google Scholar	F1 score	**0.79**	**0.68**	**0.47**	0.73	0.55	0.23
	Time (m)	5.92	7.04	8.57	2.51	2.6	6.51

Table 5.3: Results of learning over all datasets with MDs and CFD violations. p is the percentage of CFD violation.

5.2.3 Impact of Number of Iterations

We have used the values 3, 4, and 5 for the number of iterations, d, for DBLP+Google Scholar, IMDb+OMDb, and Walmart+Amazon datasets, respectively. Table 5.4 shows data regarding the scalability of DLearn-CFD over IMDb+OMDb (3 MD + 4 CFD). The largest contributor to runtime for the DLearn system is the number of iterations (d) that the system is configured to run during the bottom clause generating phase. A higher d-value increases both the effectiveness as well as the runtime. A d-value higher than 4 generates a very modest increase in effectiveness with a substantial increase in runtime. This result indicates that for a given dataset, the learning algorithm can access most relevant tuples for a reasonable value of d. Increasing the d-value beyond that point will not significantly increase the effectiveness.

Metric	$k_m = 5$			
	d=2	d=3	d=4	d=5
F-1 Score	0.52	0.52	0.78	0.80
Time (m)	1.35	4.35	16.26	37.56

Table 5.4: Results of changing the number of iterations.

5.2.4 Scalability of DLearn

We evaluate the effect of the number of training examples on both DLearn's effectiveness and efficiency. We use the IMDb + OMDb (three MDs) dataset and fix $k_m = 2$. We generate 2,100 positive and 4,200 negative examples. From these sets, we use 100 positive and 200 negative examples for testing. From the remaining examples, we generate

training sets containing 100, 500, 1,000, and 2,000 positive examples, and we double the number of negative examples. For each training set, we use DLearn with MD support to learn a definition. Figure 5.1 (left) shows the F1 scores and learning times for each training set. With 100 positive and 200 negative examples, DLearn obtains an F1 score of 0.80. With 500 positive and 1000 negative examples, the F1 score increases to 0.91. More training examples do not affect the F1 score significantly. On the other hand, the learning time consistently increases with the number of training examples. Nevertheless, DLearn can learn efficiently even with the largest training set. We also evaluate DLearn with support for both MDs and CFDs' violations and report the results in Table 5.5. It indicates that DLearn with CFD and MD support can deliver effective results efficiently over many examples with $k_m = 2$.

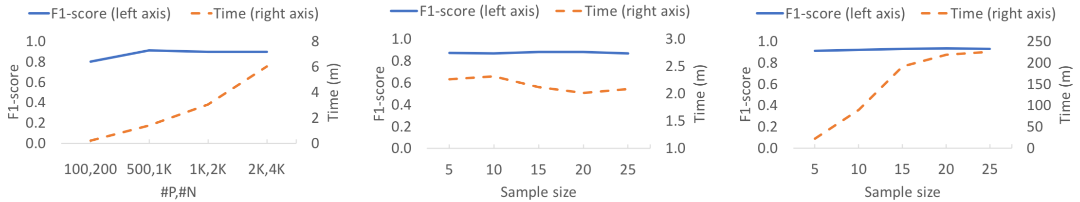

Figure 5.1: Learning over the IMDb+OMDb (3 MDs) dataset while increasing the number of positive and negative (#P, #N) examples (left) and while increasing sample size for $k_m = 2$ (middle) and $k_m = 5$ (right).

#P/#N	$k_m = 5$				$k_m = 2$			
	100/200	500/1k	1k/2k	2k/4k	100/200	500/1k	1k/2k	2k/4k
F1 score	0.78	0.82	0.81	0.82	0.78	0.79	0.81	0.81
Time (m)	16.26	72.16	121.04	317.5	0.34	2.01	2.76	5.19

Table 5.5: Learning over the IMDb+OMDb (3 MDs) with CFD violations by increasing positive (#P) and negative (#N) examples.

5.2.5 Effect of Sampling

Next, we evaluate the effect of sampling on DLearn's effectiveness and efficiency. We use the IMDb + OMDb (three MDs) dataset and fix $k_m = 2$ and $k_m = 5$. We use 800 positive and 1,600 negative examples for training, and 200 positive and 400 negative

examples for testing. Figure 5.1 (middle and right) shows the F1 score and learning time of DLearn with $k_m = 2$ and $k_m = 5$, respectively, when varying the sample size. For both values of k_m, the F1 score does *not* change significantly with different sampling sizes. With $k_m = 2$, the learning time remains almost the same with different sampling sizes. However, with $k_m = 5$, the learning time increases significantly. Therefore, using a small sample size is enough for learning an effective definition efficiently.

Chapter 6: Conclusion and Discussion

6.1 Summary

Dirty data can negatively impact the performance of machine learning applications. Therefore, it is crucial to remove any erroneous or missing values in a dataset to ensure the accuracy of the result by leveraging unique attributes in the dataset, such as functional dependency. This thesis explores a general trend in data cleaning research, identifies a gap, and aims to provide solutions to address the unresolved challenges in data cleaning. Specifically, the research provides a foundation for preprocessing data before applying machine learning algorithms, removing the expensive and inaccurate data cleaning stage.

6.2 Limitations

The main shortcoming of this method is that relational learning requires declarative relationships between attributes, referred to as functional dependency. Unfortunately, real-life data sets are less likely to have unique declarative constraints. Even with such data constraints, domain experts with a deep understanding of the data attributes need to identify and define a datalog relation for the target, which might pose a challenge to general practitioners. In addition, the type of data that works best with relational learning is categorical, not numerical. Therefore, the range of applications might be limited to specific relational databases.

6.3 Future Research Directions

To address the limitation, we propose a probabilistic approach to infer a value based on the data distribution by replacing dirty data with a statistically representative value derived from the data set. One of the most common approaches is to use the mean [38, 37]. By integrating the statistical approach and relational learning, data cleaning can be completed efficiently and effectively with minimum effort.

Bibliography

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *The VLDB Journal*, 24:557–581, 2015.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level.* Addison-Wesley, 1994.

[3] Azza Abouzeid, Dana Angluin, Christos H. Papadimitriou, Joseph M. Hellerstein, and Abraham Silberschatz. Learning and verifying quantified boolean queries by example. In *PODS*, 2013.

[4] Foto Afrati, Chen Li, and Prasenjit Mitra. On containment of conjunctive queries with arithmetic comparisons. In *Advances in Database Technology - EDBT*, pages 459–476, 2004.

[5] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Transformation-based framework for record matching. *ICDE*, pages 40–49, 2008.

[6] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[7] Zeinab Bahmani, Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. Declarative entity resolution via matching dependencies and answer set programs. In *KR*, 2012.

[8] Zeinab Bahmani, Leopoldo E. Bertossi, and Nikolaos Vasiloglou. ERBlox: Combining matching dependencies with machine learning for entity resolution. In *SUM*, 2015.

[9] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal*, 18:255–276, 2008.

[10] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory of Computing Systems*, 52:441–482, 2011.

[11] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 746–755, 2007.

[12] 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*, page 143?154, New York, NY, USA, 2005. Association for Computing Machinery.

[13] Douglas Burdick, Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang-Chiew Tan. A declarative framework for linking entities. *ACM Trans. Database Syst.*, 41(3):17:1–17:38, 2016.

[14] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD Conference*, 2016.

[15] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. pages 315–326, 01 2007.

[16] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, and Pradap Konda. The Magellan data repository. `https://sites.google.com/site/anhaidgroup/projects/data`.

[17] Luc De Raedt. *Logical and Relational Learning*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[18] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[19] Pedro Domingos. Machine learning for data management: Problems and solutions. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD ?18, page 629, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Trans. Database Syst.*, 22:364–418, 1997.

[21] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018.

[22] Wenfei Fan. Dependencies revisited for improving data quality. In Maurizio Lenzerini and Domenico Lembo, editors, *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 159–170. ACM, 2008.

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

[24] Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. Reasoning about record matching rules. *PVLDB*, 2:407–418, 2009.

[25] Enrico Franconi1, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census data repair: A challenging application of disjunctive logic programming. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 561–578, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

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

[27] Hector GarciaMolina, Jeff Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008.

[28] Lise Getoor and Ashwin Machanavajjhala. Entity resolution for big data. In *KDD*, 2013.

[29] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[30] 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.

[31] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162 3:705–708, 1982.

[32] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *SIGMOD Conference*, 1995.

[33] Miguel Ángel Hernández, Georgia Koutrika, Rajasekar Krishnamurthy, Lucian Popa, and Ryan Wisnesky. HIL: a high-level scripting language for entity integration. In *EDBT*, 2013.

[34] Ihab F. Ilyas. Effective data cleaning with continuous evaluation. *IEEE Data Eng. Bull.*, 39(2):38–46, 2016.

[35] Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. Fastqre: Fast query reverse engineering. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 337–350, New York, NY, USA, 2018. ACM.

[36] Bojan Karlaš, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. Nearest neighbor classifiers over incomplete information: From certain answers to certain predictions, 2020.

[37] Pasha Khosravi, YooJung Choi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. On tractable computation of expected predictions. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[38] Pasha Khosravi, Yitao Liang, YooJung Choi, and Guy Van den Broeck. What to expect of classifiers? reasoning about logistic regression with missing features. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 2716–2724. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[39] Angelika Kimmig, David Poole, and Jay Pujara. Statistical relational ai (starai) workshop. In *AAAI*, 2020.

[40] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*, page 53?62, New York, NY, USA, 2009. Association for Computing Machinery.

[41] Ioannis Koumarelas, Thorsten Papenbrock, and Felix Naumann. Mdedup: Duplicate detection with matching dependencies. *Proceedings of the VLDB Endowment*, 13(5):712–725, 2020.

[42] S. Krishnan, Jiannan Wang, M. Franklin, Ken Goldberg, Tim Kraska, T. Milo, and Eugene Wu. Sampleclean: Fast and reliable analytics on dirty data. *IEEE Data Eng. Bull.*, 38:59–75, 2015.

[43] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Kenneth Y. Goldberg. ActiveClean: Interactive data cleaning for statistical modeling. *PVLDB*, 9:948–959, 2016.

[44] Sanjay Krishnan and Eugene Wu. Alphaclean: Automatic generation of data cleaning pipelines. 2019.

[45] Ni Lao, Einat Minkov, and William Cohen. Learning relational features with backward random walks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 666–675, Beijing, China, July 2015. Association for Computational Linguistics.

[46] Ga Young Lee, Lubna Alzamil, Bakhtiyar Doskenov, and Arash Termehchy. A survey on data cleaning methods for improved machine learning model performance. *CoRR*, abs/2109.07127, 2021.

[47] Hao Li, Chee Yong Chan, and David Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8:2158–2169, 2015.

[48] Lilyana Mihalkova and Raymond J. Mooney. Bottom-up learning of Markov logic network structure. In *ICML*, 2007.

[49] Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[50] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *ALT*, 1990.

[51] Stephen Muggleton, Jose Santos, and Alireza Tamaddoni-Nezhad. ProGolem: A system based on relative minimal generalisation. In *ILP*, 2009.

[52] Jose Picado, John Davis, Arash Termehchy, and Ga Young Lee. *Learning Over Dirty Data Without Cleaning*, page 1301–1316. Association for Computing Machinery, New York, NY, USA, 2020.

[53] Jose Picado, Arash Termehchy, and Alan Fern. Schema independent relational learning. In *SIGMOD Conference*, 2017.

[54] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[55] De Raedt. Logical and Relational Learning. *Springer Publishing Company, Incorporated*, 2010.

[56] Luc De Raedt, David Poole, Kristian Kersting, and Sriraam Natarajan. Statistical relational artificial intelligence: Logic, probability and computation. In *NeurIPS*, 2017.

[57] Theodoros I. Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. HoloClean: Holistic data repairs with probabilistic inference. *PVLDB*, 10:1190–1201, 2017.

[58] Matthew Richardson and Pedro M. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[59] Melanie Weis, Felix Naumann, Ulrich Jehle, Jens Lufter, and Holger Schuster. Industry-scale duplicate detection. *PVLDB*, 1:1253–1264, 2008.

[60] Jef Wijsen. Condensed representation of database repairs for consistent query answering. In *Proceedings of the 9th International Conference on Database Theory*, page 378?393, Berlin, Heidelberg, 2003. Springer-Verlag.

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

[62] Qiang Zeng, Jignesh M. Patel, and David Page. QuickFOIL: Scalable inductive logic programming. *PVLDB*, 8:197–208, 2014.