AN ABSTRACT OF THE DISSERTATION OF

Jose Manuel Picado Leiva for the degree of Doctor of Philosophy in Computer Science
presented on May 28, 2019.

Title: Representationally Robust and Scalable Learning over Relational Databases

Abstract approved: ____

Arash Termehchy

Learning novel concepts from relational databases is an important problem with applications in several disciplines, such as data management, natural language processing, and bioinformatics. For a learning algorithm to be effective, the input data should be clean and in some desired representation. However, real-world data is usually heterogeneous – the same data may be represented under different representations. The current approach to effectively use learning algorithms is to find the desired representations for these algorithms, transform the data to these representations, and clean the data. These tasks are hard and time-consuming and are major obstacles for unlocking the value of data. This thesis demonstrates that it is possible to develop robust learning algorithms that learn in the presence of representational variations. We develop two systems called Castor and CastorX, which exploit data dependencies to be robust against different types of representational variations. Further, we propose several techniques that allow these systems to learn efficiently over large databases. The proposed systems learn over the original data, removing the need for transforming the data before applying learning algorithms. Our results show that Castor and CastorX learn accurately and efficiently over real-world databases. This work paves the way for new approaches that replace pre-processing tasks such as data wrangling with robust learning algorithms.

©Copyright by Jose Manuel Picado Leiva
May 28, 2019
All Rights Reserved

Representationally Robust and Scalable Learning over Relational Databases

by

Jose Manuel Picado Leiva

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented May 28, 2019
Commencement June 2019

Doctor of Philosophy dissertation of Jose Manuel Picado Leiva presented on May 28, 2019.

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

Jose Manuel Picado Leiva, Author

ACKNOWLEDGEMENTS

I would like to thank my major advisor, Arash Termehchy. Arash taught me to work on visionary and potentially transformative research projects. He gave me the freedom to explore different areas, while at the same time making sure that I stayed focused on making progress towards my degree. I will forever be grateful for his patient and driving guidance throughout this long and enriching journey.

This thesis would not have been possible without the work of multiple people. I would like to acknowledge my co-authors Arash Termehchy, Alan Fern, Sudhanshu Pathak, and Parisa Ataei. Their ideas and hard work shaped the research and developments presented in this thesis. I would like to thank Peter Samouelian for using Castor and providing ideas for improving it. I would also like to thank the members of my committee: Alan Fern, David Maier, Prasad Tadepalli, and Hector Vergara. Their insightful comments helped this thesis become a better product.

During my graduate studies, I worked with and learned from many fellow graduate students. I would like to acknowledge the members of the IDEA Lab, and in particular, the following colleagues with whom I worked and shared ideas over the years: Vahid, Sudhanshu, Ben, and Yods. Attending graduate school along with fellow graduate students made the journey enjoyable and intellectually stimulating. I would also like to thank the Department of Electrical Engineering and Computer Science for giving me the opportunity to pursue a doctoral degree at Oregon State University. The entire faculty and staff provided invaluable education and made this experience very gratifying.

During my time at Corvallis, I was fortunate enough to meet awesome people and make great friends. I would like to thank all of them for making this an amazing experience. Finally, I would like to thank my family. They provided the bases for who I am today. They encouraged me to follow my dreams and supported me during all these years. Thank you for all your infinite love and support.

TABLE OF CONTENTS

Page

1 Introduction 1

1.1 Objectives 2

1.2 Technical Contributions 2

2 Preliminaries 6

2.1 Basic Definitions 6

2.2 Relational Learning 7

3 Robustness Against Structural Heterogeneities 9

3.1 Motivation 9

3.2 Framework for Schema Independence 12

3.2.1 Relational Learning 12

3.2.2 Mapping Database Instances 13

3.2.3 Mapping Definitions 15

3.2.4 Bijective and Definition-Bijective Transformations 15

3.2.5 Schema Independence Property 18

3.3 Composition and Decomposition 19

3.4 Top-down Algorithms 21

3.5 Bottom-up Algorithms 23

3.5.1 Bottom-clause Construction 23

3.5.2 Golem 26

3.5.3 ProGolem 30

3.6 Castor 31

3.6.1 Castor Bottom-Clause Construction 32

3.6.2 Castor Generalization 37

3.6.3 Generating Safe Clauses 43

3.6.4 General Composition and Decomposition 44

3.6.5 Castor System Design Choices and Implementation 48

3.7 Query-based algorithms 49

3.8 Experiments 52

3.8.1 Experimental Settings 53

3.8.2 Sample-based Algorithms 60

3.8.3 Effect of Castor Design Choices 67

3.8.4 Query-based Algorithms 69

TABLE OF CONTENTS (Continued)

Page

3.9 Related Work 70

4 Automatically Setting Language Bias 73

4.1 Motivation 73

4.2 Language Bias 75

4.3 AutoMode System 76
4.3.1 Generating Predicate Definitions 76
4.3.2 Generating Mode Definitions 79

4.4 Improving Efficiency Through Sampling 80
4.4.1 Naïve Sampling 81
4.4.2 Random Sampling 81
4.4.3 Stratified Sampling 82

4.5 Empirical Results 83
4.5.1 Evaluating AutoMode 84
4.5.2 Evaluating Sampling Techniques 86

5 Robustness Against Content Heterogeneities 89

5.1 Motivation 89

5.2 Matching & Cleaning 93
5.2.1 Matching Dependencies 93
5.2.2 Stable Instances 95

5.3 Learning Over Heterogeneous Data 97
5.3.1 Approaches to Learning Over Heterogeneous Data 97
5.3.2 Representing Heterogeneity in Definitions 98
5.3.3 Coverage Over Heterogeneous Data 100

5.4 CastorX: A Learning Algorithm for Heterogeneous Data 102
5.4.1 Bottom-clause Construction 103
5.4.2 Generalization 105
5.4.3 Commutativity Property 112
5.4.4 Using the Learned Definitions 116

5.5 Implementation Details 116
5.5.1 Matching Dependencies 117

5.6 Experiments 117
5.6.1 Experimental Settings 118

TABLE OF CONTENTS (Continued)

Page

5.6.2 Effectiveness and Efficiency . . . 121
5.6.3 Scalability of CastorX . . . 124
5.6.4 Effect of Sampling . . . 124

5.7 Related Work . . . 126

6 Conclusion and Future Work 128

6.1 Summary . . . 128

6.2 Future Work . . . 129

Bibliography 130

LIST OF FIGURES

Figure | Page

3.1 Effect of parallelization on Castor's running time over the HIV-Large dataset. 67

3.2 Effect of parallelization on Castor's running time over the HIV-2K4K dataset. 68

3.3 Effect of parallelization on Castor's running time over the IMDb dataset. 68

3.4 Average number of equivalence queries by the A2 algorithm. 71

3.5 Average number of membership queries by the A2 algorithm. 71

4.1 Architecture of the AutoMode system. 76

4.2 A fragment of the type graph for the UW-CSE dataset. Solid lines represent exact INDs and dashed lines represent approximate INDs. 79

5.1 Results of learning over the IMDB+OMDB (three MDs) dataset while increasing the number of positive and negative (#P, #N) training examples. 125

5.2 Results of learning over the IMDB+OMDB (three MDs) dataset while increasing sample size for $k_m = 2$. 125

5.3 Results of learning over the IMDB+OMDB (three MDs) dataset while increasing sample size for $k_m = 5$. 126

LIST OF TABLES

Table Page

1.1 Schema for the UW-CSE dataset. 3

3.1 Two schemas for the UW-CSE dataset. 10

3.2 Example database over the 4NF schema of the UW-CSE dataset. 25

3.3 Numbers of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset. 54

3.4 Schemas for the HIV-Large and HIV-2K4K datasets. 54

3.5 INDs in the initial HIV dataset. 54

3.6 INDs in the UW-CSE dataset. Top: INDs in the original dataset. Middle: INDs added to have bijective transformations. Bottom: INDs that should hold according to the semantics of the database. 55

3.7 JMDB and Stanford schemas for the IMDb dataset. Relations in bottom are contained in both schemas. 56

3.8 Denormalized schema for the IMDb dataset. 57

3.9 INDs in IMDb dataset. 57

3.10 Results of learning relations over the HIV-Large and HIV-2K4K dataset. . 62

3.11 Results of learning relations over the UW-CSE dataset. 63

3.12 Results of learning relations over the IMDb dataset. 63

3.13 Results of learning over the HIV-2K4K, UW-CSE and IMDb datasets using INDs in the form of subset. 65

3.14 Impact of stored procedures on Castor's running time over the HIV-Large, HIV-2K4K, and IMDb datasets. 69

4.1 Schema for the UW-CSE dataset. 73

4.2 A subset of predicate and mode definitions for the UW-CSE dataset. . . 74

4.3 Number of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset. 84

LIST OF TABLES (Continued)

Table | Page

4.4 Results of learning relations over UW-CSE, HIV, and IMDb data (h=hours, m=minutes, s=seconds). 87

4.5 New schema for the HIV dataset. 87

4.6 Results of learning relations over HIV data with different sampling techniques (h=hours, m=minutes). 88

5.1 Schema fragments for the IMDb and Box Office Mojo datasets. 89

5.2 Example movie database. 104

5.3 Numbers of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset. 118

5.4 Results of learning over all datasets. Number of top similar matches denoted by k_m. 122

LIST OF ALGORITHMS

Algorithm		Page
1	Generic relational learning algorithm following a covering approach.	8
2	Bottom-clause construction algorithm.	24
3	Golem's *LearnClause* algorithm.	28
4	ProGolem's *ARMG* algorithm.	30
5	Castor's *LearnClause* algorithm.	33
6	Castor's negative reduction algorithm.	41
7	Algorithm to generate the type graph.	78
8	Bottom-clause construction algorithm using random sampling.	82
9	Bottom-clause construction algorithm using stratified sampling.	83
10	CastorX's bottom-clause construction algorithm.	105

Chapter 1: Introduction

Learning novel concepts and relations from relational databases is an important problem with applications in several disciplines, such as data management, natural language processing, and bioinformatics [22, 34, 60]. For a learning algorithm to be effective, the input data should be clean and in some desired representation. Therefore, to use current learning algorithms, users have to find the desired representations for these algorithms, transform their data to these representations, and clean the data. This thesis demonstrates that it is possible to develop robust learning algorithms that learn in the presence of representational variations in the data by exploiting data dependencies.

One example of a scenario where robust learning algorithms are useful is when using data from two or more data sources to learn a novel concept. Different data sources may use different names to refer to the same entity. For example, one database may contain the value *J. Smith*, while another database may contain the value *John Smith*, even though they refer to the same entity. The use of different names to refer to the same entity is an example of a *representational variation*. One approach to solve the problem of representational variations is to resolve the heterogeneities in the data to create a unified and clean database instance to be used for learning. However, this process requires a great deal of time and effort. Further, it is not usually clear which values represent the same real-world entity in different databases. For example, besides containing the value *John Smith*, the second database may also contain the value *Jeremy Smith*. It is not clear whether the value *J. Smith* in the first database refers to *John Smith* or *Jeremy Smith* in the second database. Instead of resolving the heterogeneities in a pre-processing step, a user may provide a set of declarative constraints that specify the attributes across relations or databases that contain values that may refer to the same real-world entity. Taking these constraints as input, a robust learning algorithm should be able to explore all possible matchings and learn an effective definition.

1.1 Objectives

In this work, we develop learning algorithms that are robust to representational variations. We seek, through robust learning algorithms, to reduce some of the obstacles that users face when learning in the presence of representational variations. We propose to use techniques from the database literature to develop robust and efficient learning algorithms that learn novel concepts over relational databases. In particular, we integrate data dependencies, such as inclusion dependencies (INDs) [2] and matching dependencies (MDs) [30], into the learning algorithms. These dependencies are specified by the user though declarative constraints or can be discovered from the database. They contain useful information about the structure and content of the data that can be exploited by the learning algorithms.

We focus on supervised learning algorithms that learn directly from relational databases. Given a relational database and training examples for a new target relation, *relational machine learning* (relational learning) algorithms learn an (approximate) relational definition of the target relation in terms of existing relations in the database [22, 34]. Learned definitions are usually first-order logic formulas and often restricted to Datalog programs. For example, consider the UW-CSE database (*alchemy.cs.washington.edu/data/uw-cse*), which contains information about a computer science department. Its schema fragments are shown in Table 1.1. Given the UW-CSE database and a set of student-advisor pairs, one may want to predict the new relation *advisedBy(stud, prof)*, which indicates that the student *stud* is advised by professor *prof*. A relational learning algorithm may learn the following Datalog program for the *advisedBy* relation:

$$advisedBy(x, y) \leftarrow publication(z, x), publication(z, y),$$

which indicates that a student is advised by a professor if they have been co-authors of a publication. Some benefits of relational learning algorithms are that they can exploit the relational structure of the data and that their learned definitions are interpretable and easy to understand.

1.2 Technical Contributions

The following is a summary of the technical contributions presented in this dissertation.

student(stud)	professor(prof)
inPhase(stud, phase)	hasPosition(prof, position)
yearsInProgram(stud, years)	taughtBy(course, prof, term)
courseLevel(course, level)	ta(course, stud, term)
publication(title, person)	

Table 1.1: Schema for the UW-CSE dataset.

Robustness Against Structural Heterogeneities. It is well established that the same relational database may be represented under different schemas for various reasons, such as efficiency, data quality, and usability. We refer to this representational variation as *structural heterogeneity*. Learning algorithms that learn over structured data, such as relational databases, must employ heuristics to learn efficiently. Structural heterogeneities present a problem for these algorithms, as their output tends to vary quite substantially over the choice of schema. In Chapter 3, we introduce the property of *schema independence* for relational learning algorithms, and study both the theoretical and empirical dependence of existing algorithms on the common class of (de-)composition schema transformations. We show theoretically and empirically that current relational learning algorithms are generally not schema independent. We propose *Castor* [54, 55], a relational learning algorithm that is schema independent. Castor achieves schema independence by leveraging data constraints called inclusion dependencies. We support the theoretical results with an empirical study that demonstrates the schema dependence or independence of several algorithms on existing benchmark and real-world datasets under (de-)composition transformations.

Automatically Setting Language Bias. Relational learning algorithms learn over relational domains, which generally contain multiple entities and relationships between entities. This high expressibility comes at the expense of a large hypothesis space. In order to constrain the hypothesis space, users must tune the learning algorithms. The *language bias* of relational learning algorithms restricts the structure and syntax of the learned Datalog programs. Unfortunately, specifying the language bias is done via trial and error and is guided by experts' intuitions. Hence, it normally takes a great deal of time and effort to effectively use these algorithms. In particular, it is hard to find a user that knows computer science concepts, such as database schema, understands the learning algorithms, and has a reasonable intuition about the target relation in special

domains. Further, it is difficult to specify the language bias for databases with large and complex schemas. Also, the language bias has to be rewritten every time that the schema changes. In Chapter 4, we propose *AutoMode* [53], a system that leverages information in the schema and content of the database, encoded in data dependencies, to automatically induce the language bias used by relational learning systems such as Castor. We show that AutoMode delivers the same accuracy as using manually written language bias by imposing only a slight overhead on the running time of the learning algorithm.

Because of their high expressibility, relational learning algorithms do not generally scale to large databases – databases with dozens of relations and thousands of tuples. Further, AutoMode may generate a language bias that does not restrict hypothesis spaces enough to learn over large databases. In Chapter 4, we also propose to use sampling techniques to get a subset of the data that is used to generate candidate definitions of the target relation. We study different sampling techniques and integrate them into the algorithm that builds candidate definitions in Castor. We show that the effectiveness and efficiency of the learning algorithms can improve with the appropriate sampling techniques.

Robustness Against Content Heterogeneities. One form of representational variation consists in using different names to refer to the same entity. This form of representational variation is particularly common when the information about a domain is spread across several data sources or when the input data has low quality. We refer to this representational variation as *content heterogeneity*. Content heterogeneity is problematic because learning algorithms treat database entities with different names as different real-world entities. This issue may significantly impact the accuracy of the learned models. In Chapter 5, we propose *CastorX* [56], an extension of Castor that performs relational learning over heterogeneous databases. CastorX takes as input declarative constraints called *matching dependencies*. Matching dependencies specify the attributes across relations or databases that contain values that may refer to the same real-world entity. CastorX encodes the information about the heterogeneity of the data, specified through matching dependencies, in the candidate definitions of the target relation. CastorX learns a definition over the heterogeneous databases, and then integrates some of the databases only if the learned definition so requires. That is, CastorX interchanges the integration and learning operations. We show that CastorX is able to learn accurate definitions over heterogeneous databases efficiently.

We conclude with Chapter 6 where we present a summary of the achievements and ideas for future directions.

Chapter 2: Preliminaries

In this chapter, we introduce some background concepts related to relational databases and first-order logic. These concepts are used throughout all chapters of this dissertation. Next, we explain relational machine learning algorithms, which we use to learn concepts over relational databases.

2.1 Basic Definitions

We fix two disjoint (countably) infinite sets of relation symbols and attribute symbols. Each relation symbol R is associated with a set of attribute symbols denoted as $sort(R)$. Let D be a countably infinite domain of values, i.e., constants. An instance I_R of relation symbol R with $n = |sort(R)|$ is a (finite) relation over D^n. The number of attributes in R, i.e., $|sort(R)|$, is called the *arity* of R. A *schema* $\mathcal{R}$ is a pair $(\mathbf{R}, \Sigma)$, where $\mathbf{R}$ and Σ are finite sets of relation symbols and constraints, respectively.

A *constraint* restricts the properties of data stored in a database. Examples of constraints are *functional dependencies* (FD) and *inclusion dependencies* (IND), i.e., referential integrity. We also refer to constraints as *data dependencies*. Let $\pi_X(I_R)$, where $X \subseteq sort(R)$, denote the projection of relation I_R on attribute set X. Let $\pi_X(t)$, where $t \in I_R$ and $X \subseteq sort(R)$, denote the projection of tuple t on attribute set X. Relation I_R satisfies FD $X \rightarrow Y$, where $X, Y \subset sort(R)$, if for each pair s, t of tuples in I_R, $\pi_X(s) = \pi_X(t)$ implies $\pi_Y(s) = \pi_Y(t)$. Given relation symbols R and S and sets of attributes $X \in sort(R)$ and $Y \in sort(S)$, relations I_R and I_S satisfy IND $R[X] \subseteq S[Y]$ if $\pi_X(I_R) \subseteq \pi_Y(I_S)$. If both INDs $R[X] \subseteq S[Y]$ and $S[Y] \subseteq R[X]$ hold in a schema, we denote them as $R[X] = S[Y]$ and call it an *IND with equality*. An *instance* of schema $\mathcal{R}$ is a mapping I over $\mathcal{R}$ that associates each relation $R \in \mathcal{R}$ to an instance I_R that satisfies all constraints in Σ. The set Σ may logically imply other constraints, e.g., FDs $X \rightarrow Y$ and $Y \rightarrow Z$ imply $X \rightarrow Z$ [2]. The set of all constraints implied by Σ is denoted by Σ^+. To simplify our notations, we use Σ and Σ^+ interchangeably.

An *atom* is a formula in the form of $R(u_1, \ldots, u_n)$, where R is a relation symbol, $n =$

$|sort(R)|$, and each u_i, $1 \leq i \leq n$, is a variable or constant. If all u_i's are constants, the atom is a *ground atom*. A *literal* is an atom or the negation of an atom. A *ground literal* is a literal whose atom is a ground atom. A *clause* is a finite disjunction of literals, where at least one literal is positive, i.e., unnegated. A *definite Horn clause* (Horn clause for short) is a clause with exactly one positive literal. The positive literal is called the head of the clause, and the set of negative literals is called the body. A Horn clause has the form: $T(\mathbf{u}) \leftarrow L_1(\mathbf{u_1}), \cdots, L_n(\mathbf{u_n})$. A Horn clause is *non-recursive* if its body only contains literals with relation symbols different from the head literal. Horn clauses are also called *Datalog rules* or *conjunctive queries* [2]. A *Horn definition* is a set of Horn clauses with the same head literal. Horn definitions are also called *Datalog programs* or *unions of conjunctive queries*. A Horn definition is non-recursive if it only contains non-recursive clauses. A Horn definition is defined over schema $\mathcal{R}$ if the bodies of all clauses in the definition contain only literals whose relation schemas are in $\mathcal{R}$. A literal L_i in a clause $H \leftarrow L_1, \ldots, L_n$ is *head-connected* if and only if at least one variable in every L_i appears either in H or in a body literal L_j, where $1 \leq j < i$.

Clause C *θ-subsumes* clause C', denoted by $C \subseteq_\theta C'$, if and only if there is some substitution θ such that $C\theta \subseteq C'$ [2, 22]. $C\theta \subseteq C'$ means that the result of applying substitution θ to clause C is a subset of clause C'. The θ-subsumption framework is both sound and complete for Horn clauses without functions [2].

2.2 Relational Learning

Given a relational database instance I and training examples E for a new target relation T, *relational machine learning* (relational learning) algorithms learn an (approximate) relational definition of T in terms of existing relations in the database [22, 34]. Training examples E are usually tuples of a single target relation T, which express positive (E^+) or negative (E^-) examples. The input database instance I is also called *background knowledge*. The learned definition H is called a *hypothesis*. For efficiency reasons, hypotheses are usually restricted to Horn definitions without negation. A learned clause C *covers* an example e if $I \wedge C \models e$, where $\models$ is the entailment operator, i.e., if I and C are true, then e is true. Definition H covers an example e if any of the clauses in H covers e.

One benefit of relational learning algorithms is that they can exploit the relational

structure of the data. Moreover, their learned definitions are interpretable and easy to understand. Relational learning has several applications in database management and machine learning, such as learning database queries [3], learning new features [21], and learning the structure of statistical relational learning (SRL) models [33, 34].

Algorithm 1: Generic relational learning algorithm following a covering approach.

Input : Database instance I, positive examples E^+, negative examples E^-
Output: A Horn definition H

```
H = {}
U = E+
while U is not empty do
    C = LearnClause(I, U, E-)
    if C satisfies minimum condition then
        H = H ∪ C
        U = U − {c ∈ U | I ∪ H ⊨ c}
return H
```

Relational learning algorithms generally follow a covering approach [47, 48, 50, 54, 57]. The covering approach consists in constructing one clause at a time. After building a clause, the algorithm adds the clause to the hypothesis, removes the positive examples covered by the clause, and moves on to learn a new clause. Algorithm 1 sketches a generic relational learning algorithm that follows a covering approach. The strategy followed by the *LearnClause* function depends on the nature of the algorithm. In top-down algorithms, the *LearnClause* function searches the hypothesis space from general to specific hypotheses. In bottom-up algorithms, the *LearnClause* function searches the hypothesis space from specific to general hypotheses. In the following chapters, we provide concrete definitions of several relational learning algorithms.

Chapter 3: Robustness Against Structural Heterogeneities

3.1 Motivation

Learning novel concepts over relational databases has attracted a great deal of attention due to its applications in data management and machine learning [22, 60, 34]. Given a relational database and training examples for some target relation, *relational machine learning* (relational learning) algorithms attempt to learn (approximate) relational definitions of the target relation in terms of existing relations in the database [22, 54, 67]. Since the space of possible definitions, e.g., all Datalog programs, is enormous, relational learning algorithms must employ heuristics to search for effective definitions. Unfortunately, such heuristics typically depend on the precise choice of schema of the underlying database. This issue occurs even if the schemas represent essentially the same information. For example, consider the UW-CSE database (*alchemy.cs.washington.edu/data/uw-cse*), which contains information about a computer science department. Table 3.1 shows two schemas for the UW-CSE database, which is used as a common relational learning benchmark. The Original schema was designed by relational learning experts. This design is generally discouraged in the database community, as it delivers poor usability and performance in query processing without providing any advantages in terms of data quality in return [2]. A database designer may use a schema closer to the 4NF schema in Table 3.1. Because each student *stud* has only one *phase* and *years*, a database designer may compose relations *student*, *inPhase*, and *yearsInProgram*. She may also combine relations *professor* and *hasPosition*. Such schema is more understandable and has shorter query execution times, without introducing any redundancy.

Example 3.1.1. *We use the classic relational learning algorithm FOIL [57] to learn a definition for the advisedBy(stud, prof) relation over the Original and 4NF schemas of the UW-CSE database, shown in Table 3.1. FOIL learns the following Datalog rule over the UW-CSE database with the Original schema:*

$$advisedBy(x, y) \leftarrow inPhase(x, \text{`}post_quals\text{'}), ta(v, x, w), taughtBy(v, y, z).$$

Original Schema	4NF Schema
student(<u>stud</u>)	student(<u>stud</u>,phase,years)
inPhase(<u>stud</u>,phase)	professor(<u>prof</u>,position)
yearsInProgram(<u>stud</u>,years)	publication(<u>title</u>,<u>person</u>)
professor(<u>prof</u>)	courseLevel(<u>crs</u>,level)
hasPosition(<u>prof</u>,position)	taughtBy(<u>crs</u>,<u>prof</u>,<u>term</u>)
publication(<u>title</u>,<u>person</u>)	ta(<u>crs</u>,<u>stud</u>,<u>term</u>)
courseLevel(<u>crs</u>,level)	
taughtBy(<u>crs</u>,<u>prof</u>,<u>term</u>)	
ta(<u>crs</u>,<u>stud</u>,<u>term</u>)	

Table 3.1: Two schemas for the UW-CSE dataset.

On the other hand, FOIL learns the following Datalog rule over the UW-CSE database with the 4NF schema:

$$advisedBy(x, y) \leftarrow student(x, \text{`}post_generals\text{'}, v), publication(z, x), publication(z, y).$$

The testing set contains 9 positive examples and 18 negative examples. The rule learned over the Original schema covers 1 positive example and 0 negative examples. On the other hand, the rule learned over the 4NF schema covers 4 positive examples and 1 negative example.

Generally, there is no canonical schema for a particular set of content in practice and people often represent the same information in different schemas [2, 29]. For example, it is generally easier to enforce integrity constraints over highly normalized schemas [2]. On the other hand, because more-normalized schemas usually contain many relations, they are hard to understand and maintain. It also takes a relatively long time to answer queries over database instances with such schemas [2]. Thus, a database designer may sacrifice data quality and choose a more *denormalized* schema for its data to achieve better usability or performance. Further, as the relative priorities of these objectives change over time, the schema may also evolve.

In order to effectively use relational learning algorithms, i.e., deliver definitions for the target relations that a domain expert would judge as correct and relevant, users generally have to restructure their databases. These algorithms do not normally offer any clear description of their desired schema, so database users have to rely on their

own expertise or proceed by trial and error to find such schemas. Nevertheless, we ideally want our database analytics algorithms to be used by ordinary users, not just experts who know the internals of these algorithms. Further, the structure of large-scale databases constantly evolves, and we want to move away from the need for constant expert attention to keep learning algorithms effective. Researchers often use (statistical) relational learning algorithms to solve various important core database problems, such as query learning [3], schema mapping [14], and entity resolution [33]. Thus, the issue of schema dependence appears in other areas of database management.

One approach to solving the problem of schema dependence is to run a learning algorithm over *all possible schemas* for a validation subset of the data and select the schema with the most accurate answers. Nonetheless, computing all possible schemas of a DB is generally undecidable [29]. One may limit the search space to a particular family of schemas to make their computation decidable. For instance, she may choose to check only schemas that can be transformed to other schemas via join and project operations, i.e. composition and decomposition [2]. However, the number of possible schemas within a particular family of a data set is extremely large. For example, a relational table may have an exponential number of distinct decompositions. As many learning algorithms need some time for parameter tuning under a new schema, it may take a prohibitively long time to find the best schema. Further, since relational learning algorithms need to access the content of the database, one has to transform the underlying data to the desired schema, which may not be practical for a large or constantly evolving database.

In this chapter, we introduce the property of *schema independence*, i.e., logical scalability, of relational learning algorithms. We propose a formal framework to evaluate the property of schema independence of a relational learning algorithm for a given family of schema changes. Since none of the current relational learning algorithms are schema independent, we leverage concepts from database literature to design a schema independent algorithm. The main contributions of this chapter are:

1. We define the property of schema independence (Section 3.2), which formalizes the notion of a learning algorithm returning equivalent answers over schema transformations that preserve information content.

2. We analyze the property of schema independence for the popular families of top-down [47, 57] (Section 3.4) and bottom-up [48, 50] learning algorithms (Section 3.5).

We prove that they are *not* schema independent under (de-)composition transformations.

3. We introduce Castor, a bottom-up algorithm that is provably schema independent under (de-)composition (Section 3.6). Castor achieves schema independence by integrating database constraints into the learning algorithm. Castor uses various techniques to learn efficiently over large databases.

4. We formalize the notion of schema independence for *query-based* learning algorithms, which learn the target concepts by asking queries to an *oracle*, e.g., a database user [3, 41]. We prove that algorithms in this family are not schema independent (Section 3.7).

5. We empirically compare the schema independence, effectiveness, and efficiency of Castor to some popular relational learning algorithms under (de-)composition using a widely used benchmark and real-world databases (Section 3.8). Our empirical results generally confirm our theoretical results and show that Castor is more efficient and as effective as, or more effective than, current algorithms.

3.2 Framework for Schema Independence

3.2.1 Relational Learning

Relational learning algorithms learn first-order definitions from training examples and a relational database instance I. The definitions are usually restricted to non-recursive Horn definitions without negation for efficiency reasons. In this chapter, we use relational learning algorithms to learn Horn definitions that define new target relations. The relation symbol in the head literals of all clauses in a definition is the *target relation*.

Example 3.2.1. *Consider using a relational learning algorithm and the UW-CSE database with the Original schema shown in Table 3.1 to learn a definition for the target relation* $collaborated(x, y)$*, which indicates that person* x *has collaborated with person* y*. The algorithm may return the definition*

$$collaborated(x, y) \leftarrow publication(p, x), publication(p, y),$$

which indicates that two persons have collaborated if they are co-authors.

We denote the set of all Horn definitions over schema $\mathcal{R}$ by $\mathcal{HD}_{\mathcal{R}}$. This set can be very large, which means that algorithms would need a lot of resources to explore all definitions. The *hypothesis space* of a relational learning algorithm is the set of all possible Horn definitions that the algorithm can explore. Because resources are limited in practice, algorithms accept parameters that either restrict the hypothesis space or the search strategy. For instance, an algorithm may consider only clauses whose number of literals are fewer than a given number, or may follow a greedy approach where only one clause is considered at a time. Let the *parameters* for a learning algorithm be a tuple of variables $\gamma = \langle \gamma_1, ..., \gamma_r \rangle$, where each γ_i is a parameter for the algorithm. We denote the parameter space by Γ. We denote the hypothesis space (or language) of algorithm A over schema $\mathcal{R}$ with parameters γ as $\mathcal{L}^A_{\mathcal{R},\gamma}$. The hypothesis space $\mathcal{L}^A_{\mathcal{R},\gamma}$ is a subset of $\mathcal{HD}_{\mathcal{R}}$ [47, 57].

Example 3.2.2. *Continuing Example 3.2.1, consider restricting the hypothesis space to clauses whose number of literals are fewer than a given number, which we call clause-length. Assume that we are now interested in learning a definition for the target relation collaboratedProf(x,y), which indicates that professor x has collaborated with professor y, under the Original schema. If we set clause-length* $= 5$*, the learning algorithm is able to learn the definition*

$$\mathit{collaboratedProf}(x, y) \leftarrow \mathit{professor}(x), \mathit{professor}(y), \mathit{publication}(p, x), \mathit{publication}(p, y).$$

However, if we set clause-length $= 3$*, the previous definitions is not in the hypothesis space of the algorithm.*

3.2.2 Mapping Database Instances

One may view a schema as a way of representing background knowledge used by relational learning algorithms to learn the definitions of target relations. Intuitively, in order to learn essentially the same definitions over schemas $\mathcal{R}$ and $\mathcal{S}$, we should make sure that $\mathcal{R}$ *and* $\mathcal{S}$ *represent basically the same information.* Let us denote the set of database instances of schema $\mathcal{R}$ as $\mathcal{I}(\mathcal{R})$. In order to compare the ability of $\mathcal{R}$ and $\mathcal{S}$ to represent the same information, we would like to check whether for each database instance $I \in$

$\mathcal{I}(\mathcal{R})$ there is a database instance $J \in \mathcal{I}(\mathcal{S})$ that contains basically the same information as I. We adapt the notion of equivalency between schemas to precisely state this idea [29, 40].

Given schemas $\mathcal{R}$ and $\mathcal{S}$, a *transformation* is a (computable) function $\tau : \mathcal{I}(\mathcal{R}) \to \mathcal{I}(\mathcal{S})$. For brevity, we write transformation τ as $\tau : \mathcal{R} \to \mathcal{S}$. Transformation τ is *total* if it is defined for every element of $\mathcal{I}(\mathcal{R})$. Transformation τ is *invertible* if and only if it is total and there exists a transformation $\tau^{-1} : \mathcal{S} \to \mathcal{R}$ such that the composition of τ and τ^{-1} is the identity mapping on $\mathcal{I}(\mathcal{R})$, that is $\tau^{-1}(\tau(I)) = I$ for $I \in \mathcal{I}(\mathcal{R})$. We call τ^{-1} the *inverse* of τ and say that τ is *invertible*. If transformation τ is invertible, one can convert every instance $I \in \mathcal{I}(\mathcal{R})$ to an instance $J \in \mathcal{I}(\mathcal{S})$ and reconstruct I from the available information in J. If $\tau : \mathcal{R} \to \mathcal{S}$ is *bijective*, schemas $\mathcal{R}$ and $\mathcal{S}$ are *information equivalent* via τ. Informally, if two schemas are information equivalent, one can convert the databases represented using one of them to the other without losing any information. Hence, one can reasonably argue that equivalent schemas essentially represent the same information. Our definition of information equivalence between two schemas is more restricted that the ones proposed in [29, 40]. We assume that in order for schemas $\mathcal{R}$ and $\mathcal{S}$ to be information equivalent via τ, τ^{-1} must be total. Although more restricted, this definition is sufficient to cover the transformations discussed in this chapter.

Example 3.2.3. *In addition to the functional dependencies shown in Table 3.1, let the following inclusion dependencies hold over the relations of the Original schema in this table: student[stud] = inPhase[stud], student[stud] = yearsInProgram[stud], professor[prof] = hasPosition[prof]. One may define the transformation τ that uses these inclusion dependencies to join relations student, inPhase, and yearsInPrograms in the Original schema to create relation student in the 4NF schema, and join relations professor and hasPosition in the Original schema to create relation professor in the 4NF schema. Transformation τ maps each instance of the Original schema to an instance of the 4NF schema. One may define the inverse transformation τ^{-1} that uses the functional dependencies in Table 3.1 to project relation student in the 4NF schema to relations student, inPhase and yearsInProgram in the Original schema, and project relation professor in the 4NF schema to relations hasPosition and professor in the Original schema. Transformation τ^{-1} maps each instance of the 4NF schema to an instance of the Original schema. Hence, the Original and 4NF schemas are information equivalent via transfor-*

mation τ.

3.2.3 Mapping Definitions

Let $h_{\mathcal{R}}$ and $h_{\mathcal{S}}$ be Horn definitions over schemas $\mathcal{R}$ and $\mathcal{S}$, respectively. Definitions $h_{\mathcal{R}}$ and $h_{\mathcal{S}}$ are *equivalent* if they return the same results over all corresponding database instances of $\mathcal{R}$ and $\mathcal{S}$. We use the operator $\equiv$ to indicate that two definitions are equivalent. Let $\mathcal{HD}_{\mathcal{R}}$ and $\mathcal{HD}_{\mathcal{S}}$ be the sets of all Horn definitions over schema $\mathcal{R}$ and $\mathcal{S}$, respectively. In order to learn equivalent definitions over schemas $\mathcal{R}$ and $\mathcal{S}$, we should make sure that the sets $\mathcal{HD}_{\mathcal{R}}$ and $\mathcal{HD}_{\mathcal{S}}$ contain equivalent definitions. That is, for every definition $h_{\mathcal{R}} \in \mathcal{HD}_{\mathcal{R}}$, there is a equivalent Horn definition in $\mathcal{HD}_{\mathcal{S}}$, and vice versa. If the set of Horn definitions over $\mathcal{R}$ is a superset or subset of the set of Horn definitions over $\mathcal{S}$, it is not reasonable to expect a learning algorithm to learn equivalent definitions in $\mathcal{R}$ and $\mathcal{S}$.

Let $\mathcal{L}_{\mathcal{R}}$ be a set of Horn definitions over schema $\mathcal{R}$ such that $\mathcal{L}_{\mathcal{R}} \subseteq \mathcal{HD}_{\mathcal{R}}$. Let $h_{\mathcal{R}} \in \mathcal{L}_{\mathcal{R}}$ be a Horn definition over schema $\mathcal{R}$ and $I \in \mathcal{I}(\mathcal{R})$ be a database instance. The result of applying a Horn definition $h_{\mathcal{R}}$ to database instance I is the set containing the head of all instantiations of $h_{\mathcal{R}}$ for which the body of the instantiation belongs to $\mathcal{I}(\mathcal{R})$. The result of applying $h_{\mathcal{R}}$ on I is denoted by $h_{\mathcal{R}}(I)$.

Definition 3.2.4. *Transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ is definition-preserving with respect to $\mathcal{L}_{\mathcal{R}}$ and $\mathcal{L}_{\mathcal{S}}$ if and only if there exists a total function $\delta_\tau : \mathcal{L}_{\mathcal{R}} \rightarrow \mathcal{L}_{\mathcal{S}}$ such that for every definition $h_{\mathcal{R}} \in \mathcal{L}_{\mathcal{R}}$ and $I \in \mathcal{I}(\mathcal{R})$, $h_{\mathcal{R}}(I) = \delta_\tau(h_{\mathcal{R}})(\tau(I))$.*

Intuitively, Horn definitions $h_{\mathcal{R}}$ and $\delta_\tau(h_{\mathcal{R}})$ deliver the same results over all corresponding database instances in $\mathcal{R}$ and $\mathcal{S}$, respectively. We call function δ_τ a *definition mapping* for τ. Transformation τ is *definition-bijective* with respect to $\mathcal{L}_{\mathcal{R}}$ and $\mathcal{L}_{\mathcal{S}}$ if and only if τ and τ^{-1} are definition-preserving with respect to $\mathcal{L}_{\mathcal{R}}$ and $\mathcal{L}_{\mathcal{S}}$, respectively. If τ is definition-bijective with respect to equivalent sets of Horn definitions, one can rewrite each Horn definition over $\mathcal{R}$ as an equivalent Horn definition over $\mathcal{S}$, and vice versa.

3.2.4 Bijective and Definition-Bijective Transformations

In order for a learning algorithm to learn equivalent definitions over schemas $\mathcal{R}$ and $\mathcal{S}$, where $\tau : \mathcal{R} \rightarrow \mathcal{S}$, τ should be both bijective and definition-bijective with respect

to $\mathcal{HD}_\mathcal{R}$ and $\mathcal{HD}_\mathcal{S}$. If τ is bijective, the learning algorithm takes as input the same background knowledge. If τ is definition-bijective, the hypothesis spaces of the learning algorithm over both schemas contain equivalent definitions. Nevertheless, it may be hard to check both conditions for given schemas. We extend the results by Fan and Bohannon [29] to find the relationship between the properties of bijective and definition-bijective transformations. In this chapter, we consider only transformations that can be written as sets of Horn definitions. We call these *Horn transformations*. Composition and decomposition are well-known examples of Horn transformations [2].

Example 3.2.5. *Let $\mathcal{R}$ be the Original schema and $\mathcal{S}$ be the 4NF schema in Example 3.2.3. The transformation from the Original schema to the 4NF schema can be written as the following set of Horn definitions:*

$$\begin{aligned}
\mathit{student}(x,y,z) &\leftarrow \mathit{student}(x), \mathit{inPhase}(x,y), \mathit{yearsInProgram}(x,z).\\
\mathit{professor}(x,y) &\leftarrow \mathit{professor}(x), \mathit{hasPosition}(x,y).\\
\mathit{publication}(x,y) &\leftarrow \mathit{publication}(x,y).
\end{aligned}$$

The inverse of this transformation from the 4NF to Original schema is a set of projection operators, which can also be written as the following set of Horn definitions:

$$\begin{aligned}
\mathit{student}(x) &\leftarrow \mathit{student}(x,y,z).\\
\mathit{inPhase}(x,y) &\leftarrow \mathit{student}(x,y,z).\\
\mathit{yearsInProgram}(x,z) &\leftarrow \mathit{student}(x,y,z).\\
\mathit{professor}(x) &\leftarrow \mathit{professor}(x,y).\\
\mathit{hasPosition}(x,y) &\leftarrow \mathit{professor}(x,y).\\
\mathit{publication}(x,y) &\leftarrow \mathit{publication}(x,y).
\end{aligned}$$

Let transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ and its inverse $\tau^{-1} : \mathcal{S} \rightarrow \mathcal{R}$ be Horn transformations. The head of each Horn definition in τ^{-1} is a relation in $\mathcal{R}$. Let $h_\mathcal{R}$ be a Horn definition in $\mathcal{HD}_\mathcal{R}$. Let J be in instance in $\mathcal{I}(\mathcal{S})$. The composition of $h_\mathcal{R}$ and τ^{-1} applied to J, denoted by $h_\mathcal{R} \circ \tau^{-1}(J)$, is obtained by applying transformation τ^{-1} to J, denoted by $\tau^{-1}(J)$, followed by applying definition $h_\mathcal{R}$ to $\tau^{-1}(J)$, denoted by $h_\mathcal{R}(\tau^{-1}(J))$. That is, $h_\mathcal{R} \circ \tau^{-1}(J) = h_\mathcal{R}(\tau^{-1}(J))$, for $J \in \mathcal{I}(\mathcal{S})$. Transformation τ^{-1} is a Horn transformation

whose body literals contain relations in $\mathcal{S}$. The composition $h_{\mathcal{R}} \circ \tau^{-1}$ is a Horn definition applied to the result of τ^{-1}. Therefore, the composition $h_{\mathcal{R}} \circ \tau^{-1}$ is a Horn definition that belongs to $\mathcal{HD}_{\mathcal{S}}$.

Example 3.2.6. *Consider the following Horn definition $h_{\mathcal{R}}$ defined over the Original schema of Table 3.1:*

$$\begin{aligned} \mathit{collaboratedProf}(x, y) \leftarrow & \mathit{professor}(x), \mathit{inPhase}(x, u), \mathit{professor}(y), \mathit{inPhase}(y, v), \\ & \mathit{publication}(p, x), \mathit{publication}(p, y). \end{aligned}$$

Given the transformation τ^{-1} from the 4NF schema to the Original schema given in Example 3.2.5, the composition $h_{\mathcal{R}} \circ \tau^{-1}$ is the following Horn definition:

$$\begin{aligned} \mathit{collaboratedProf}(x, y) \leftarrow & \mathit{professor}(x, u), \mathit{professor}(y, v), \\ & \mathit{publication}(p, x), \mathit{publication}(p, y). \end{aligned}$$

Proposition 3.2.7. *Given schemas $\mathcal{R}$ and $\mathcal{S}$, if transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ is bijective and both τ and τ^{-1} are Horn transformations, then τ is definition-bijective with respect to $\mathcal{HD}_{\mathcal{R}}$ and $\mathcal{HD}_{\mathcal{S}}$.*

Proof. Let us define a function $\delta_\tau : \mathcal{HD}_{\mathcal{R}} \rightarrow \mathcal{HD}_{\mathcal{S}}$ to be $\delta_\tau(h_{\mathcal{R}}) = h_{\mathcal{R}} \circ \tau^{-1}$ for any $h_{\mathcal{R}} \in \mathcal{HD}_{\mathcal{R}}$. We know that $\delta_\tau(h_{\mathcal{R}}) \in \mathcal{HD}_{\mathcal{S}}$. Furthermore, for every $h_{\mathcal{R}} \in \mathcal{HD}_{\mathcal{R}}$ and $I \in \mathcal{I}_{\mathcal{R}}$, $h_{\mathcal{R}}(I) = h_{\mathcal{R}}(\tau^{-1}(\tau(I))) = (h_{\mathcal{R}} \circ \tau^{-1})(\tau(I))) = \delta_\tau(h_{\mathcal{R}})(\tau(I))$. Similarly, we define a function $\delta'_\tau : \mathcal{HD}_{\mathcal{S}} \rightarrow \mathcal{HD}_{\mathcal{R}}$ as $\delta'_\tau(h_{\mathcal{S}}) = h_{\mathcal{S}} \circ \tau$ for any $h_{\mathcal{S}} \in \mathcal{HD}_{\mathcal{S}}$. Clearly, $\delta'_\tau(h_{\mathcal{S}}) \in \mathcal{HD}_{\mathcal{R}}$. Also, for every $h_{\mathcal{S}} \in \mathcal{HD}_{\mathcal{S}}$ and every $J \in \mathcal{I}_{\mathcal{S}}$ such that there is an $I \in \mathcal{I}_{\mathcal{R}}$ where $J = \tau(I)$, $h_{\mathcal{S}}(J) = h_{\mathcal{S}}(\tau(I)) = (h_{\mathcal{S}} \circ \tau)(I) = \delta'_\tau(h_{\mathcal{S}})(I)$. Thus, τ is definition-bijective with respect to $\mathcal{HD}_{\mathcal{R}}$ and $\mathcal{HD}_{\mathcal{S}}$. □

Intuitively, if $\tau : \mathcal{R} \rightarrow \mathcal{S}$ is bijective and both τ and τ^{-1} are Horn transformations, every Horn definition in $\mathcal{HD}_{\mathcal{R}}$ can be rewritten as a Horn definition in $\mathcal{HD}_{\mathcal{S}}$ such that they return the same results over equivalent database instances. Hence, in the rest of this chapter, we consider only the bijective Horn transformations whose inverses are Horn transformations. In this chapter, we focus on composition and decomposition transformations, explained in Section 3.3, which can be written as sets of project and join operations. There exist other types of transformations that can also be written as

Horn transformations, but are not composition and decomposition transformations. One example of such transformation is *horizontal decomposition*, where the set of tuples of one relation is decomposed into two or more relations. We leave the study of this type of transformations as future work.

Example 3.2.8. *Let $\mathcal{R}$ be the Original schema and $\mathcal{S}$ be the 4NF schema in Example 3.2.3, and $\tau : \mathcal{R} \rightarrow \mathcal{S}$ and $\tau^{-1} : \mathcal{S} \rightarrow \mathcal{R}$ are the Horn transformation explained in Example 3.2.5. According to Proposition 3.2.7, τ is definition-bijective with respect to $\mathcal{HD}_{\mathcal{R}}$ and $\mathcal{HD}_{\mathcal{S}}$.*

3.2.5 Schema Independence Property

The *hypothesis space* determines the set of possible Horn definitions that the algorithm can explore. Example 3.2.2 showed that an algorithm is able to learn a definition for a target relation with some hypothesis space but not in another, more-restricted space. In order for an algorithm to learn equivalent definitions for a target relation over schemas $\mathcal{R}$ and $\mathcal{S}$, it should have equivalent hypothesis spaces over $\mathcal{R}$ and $\mathcal{S}$. We call this property hypothesis-invariance. Let Γ be the parameter space for algorithm A.

Definition 3.2.9. *Algorithm A is hypothesis-invariant under transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ if and only if τ is definition-bijective with respect to $\mathcal{L}^{A}_{\mathcal{R},\gamma}$ and $\mathcal{L}^{A}_{\mathcal{S},\gamma}$, for all $\gamma \in \Gamma$.*

Algorithm A is hypothesis-invariant under a set of transformations if and only if A is hypothesis-invariant under every transformation in the set. We now define the notion of schema independence for relational learning algorithms over a bijective transformation. A relational learning algorithm $A(I, E, \gamma)$ takes as input a database instance I, training examples E, and parameters $\gamma \in \Gamma$, and outputs a hypothesis in $\mathcal{L}^{A}_{\mathcal{R},\gamma}$.

Definition 3.2.10. *Algorithm A is schema independent under bijective transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ if and only if A is hypothesis-invariant under τ and for every $I \in \mathcal{I}(\mathcal{R})$ and all $\gamma \in \Gamma$, we have: $A(\tau(I), E, \gamma) \equiv \delta_{\tau}(A(I, E, \gamma))$, where δ_{τ} is the definition mapping for τ.*

Algorithm A is schema independent under the set of transformations if and only if it is schema independent under each transformation in the set. Note that if an algorithm is schema independent under transformation τ, it is hypothesis-invariant under τ. However,

it is possible for an algorithm not to be schema independent, but be hypothesis-invariant. In such cases, the cause of schema dependence must necessarily be related to the search process of the algorithm, rather than hypothesis-representation capacity.

Example 3.2.11. *Consider the Original schema and the 4NF schema in Example 3.2.3. The Original schema is the result of a decomposition of the 4NF schema. Consider the learning algorithm FOIL. If the target relation is collaboratedProf(x,y), as in Example 3.2.2, FOIL is able to learn equivalent definitions under the Original schema and the 4NF schema. But, if the target relation is advisedBy(x,y), FOIL learns non-equivalent definitions under these schemas, as seen in Example 3.1.1, and is not schema independent.*

3.3 Composition and Decomposition

There are many bijective Horn transformations between relational schemas [2, 40]. In this chapter, we explore the schema independence of relational learning algorithms under two widely used Horn transformations called *decomposition*, where the transformation is the projection operator, and *composition*, where the transformation is the natural join operator [2]. Our reasons for selecting these transformations are two-fold. First, they are used in most normalizations and de-normalizations, e.g., 3rd normal form, which are arguably one of the most frequent schema modifications and their importance has been recognized from the early days of the relational model [2]. Database designers often normalize schemas to remove redundancy and insertion and deletion anomalies. On the other hand, database designers denormalize schemas to improve query processing time and schema readability [2]. We also observe several cases of them in relational learning benchmarks, one of which is presented in Section 3.1.

We define decomposition as follows [2]. Let $S_i \bowtie S_j$ and $I_{S_i} \bowtie I_{S_j}$ denote the natural join between S_i and S_j and their instances, respectively. We restrict the definition of natural join for the cases where S_i and S_j have at least one attribute symbol in common to avoid Cartesian product. Let $\bowtie_{i=1}^{n} S_i$ show the natural join between $S_1, \ldots, S_n$. Recall that if both INDs $S_1[A] \subseteq S_2[B]$ and $S_2[B] \subseteq S_1[A]$ hold in a schema, we denote them as $S_1[A] = S_2[B]$ and call it an IND with equality.

Definition 3.3.1. *A decomposition of schema* $\mathcal{R} = (\mathbf{R}, \Sigma_R)$ *with single relation sym-*

bol R is schema $\mathcal{S} = (\mathbf{S}, \Sigma_S)$ *with relation symbols* $S_1 \ldots S_n$ *such that* $sort(R) = \cup_{1 \leq i \leq n} sort(S_i)$ *and*

1. *For each relation* I_R *there is one and only one instance* $(I_{S_1} \ldots I_{S_n})$ *of* $\mathcal{S}$ *such that* $\pi_{sort(S_i)}(I_R) = I_{S_i}$, $1 \leq i \leq n$, *and* $\bowtie_{i=1}^{n} I_{S_i} = I_R$.

2. *For all* S_i, S_j, $1 \leq i, j \leq n$, *such that* $X = sort(S_i) \cap sort(S_j) \neq \emptyset$, $\Sigma_{\mathcal{S}}$ *contains IND with equality* $S_i[X] = S_j[X]$.

3. *We have* $\Sigma_S = \Sigma_R \cup \lambda$, *where* λ *is the set of INDs with equality in the second condition.*

The first and third conditions in Definition 3.3.1 are generally known as *lossless join* and *dependency preservation* properties, respectively. The second condition in Definition 3.3.1 guarantees that the natural join of relations in every instance $I_{\mathcal{S}}$ of $\mathcal{S}$ does not lose any tuples in $I_{\mathcal{S}}$. Table 3.1 depicts an example of a decomposition. Relation symbol *student* in the 4NF schema is decomposed into *student*, *inPhase*, and *yearsInProgram* in the Original schema. The conditions of Definition 3.3.1, e.g., lossless join property, hold in this example due to the functional dependencies in the Original and 4NF schemas [2]. These conditions may also be satisfied because of other types of constraints in the schema, such as multi-valued dependencies. A *composition* is the inverse of a decomposition, which is expressed by the natural join operator.

Consider again schema $\mathcal{S}$ in Definition 3.3.1. The join $\bowtie_{i=1}^{n} I_{S_i}$ is *globally consistent* if for each j, $1 \leq j \leq n$, $\pi_{sort(S_j)}(\bowtie_{i=1}^{n} I_{S_i}) = I_{S_j}$ [2]. Intuitively speaking, a join is globally consistent if none of its relations has a dangling tuple regarding the join. For example, the join between the relations of $\mathcal{S}$ in the first condition of Definition 3.3.1 is globally consistent. The join $\bowtie_{i=1}^{n} I_{S_i}$ is *pairwise consistent* if for each $1 \leq i, j \leq n$, $\pi_{sort(S_i)}(I_{S_i} \bowtie I_{S_j}) = I_{S_i}$. In other words, I_{S_i} does not lose any tuple after joining with I_{S_j}. The join $\bowtie_{i=1}^{n} S_i$ is *acyclic* if each instance $\bowtie_{i=1}^{n} I_{S_i}$ that is pairwise consistent is globally consistent [2]. For example, the join $S_1 \bowtie S_2$ in schema $\mathcal{S}_1$:$\{S_1(A,B), S_2(A,C)\}$ is acyclic. But, the join $S_3 \bowtie S_4 \bowtie S_5$ in schema $\mathcal{S}_2$: $\{S_3(A,B), S_4(B,C),\ S_5(B,A),\}$ is cyclic. In this chapter, we consider only the decompositions where the join in the first condition of Definition 3.3.1 is acyclic [2]. Acyclic joins cover most real-world decompositions [2]. For example, most normal forms, e.g., 3NF, BCNF, 4NF, have acyclic joins.

For simplicity, we consider leaving a relation unchanged as a special case of decomposition. We define a *decomposition (composition)* of a schema with more than one relation as a finite set of applications of composition and decomposition to the relations in the schema. Every decomposition is bijective [2]. Because each decomposition is bijective, every composition is also bijective. Because both projection and natural join can be written as Horn definitions, each decomposition (composition) and its inverse are Horn transformations. Hence, they are definition-bijective. In this chapter, we explore the property of schema independence for composition and decomposition.

3.4 Top-down Algorithms

In this section, we study relational learning algorithms that follow a top-down approach. The hypothesis space in top-down algorithms is a tree in which nodes represent clauses and each edge is the application of a basic refinement operator, which generally consists of adding a new literal to the clause. Top-down algorithms start from the most general clause, which corresponds to the root of the tree, and repeatedly refine it until reaching some stopping condition. The strategy of searching the tree varies between different top-down algorithms. For instance, FOIL [57, 67] is an efficient and popular top-down algorithm that follows a greedy best-first search strategy. Given the current clause, FOIL specializes a clause by adding the literal that provides the most information gain. FOIL stops adding literals to the clause when the number of bits required to encode the clause exceeds the number of bits required to indicate the number of positive examples covered by the clause. Progol [47] is another well-known top-down algorithm similar to FOIL, except that it does not follow a greedy search strategy, and it restricts the literals that can be added to the clause. Further, Progol limits the length of the clause, i.e., the maximum number of literals in a clause.

Intuitively, because composition and decomposition modify the number of relations in a schema, equivalent clauses over the original and transformed schemas may have different lengths and would require different number of bits to be encoded. Hence, the stopping conditions used by FOIL and Progol may produce different hypothesis spaces over different schemas.

Theorem 3.4.1. *FOIL is not hypothesis-invariant.*

Proof. Let $\mathcal{R}$ be a schema, E be the training data, C be a clause, n be the number of variables in C, and p be the number of positive examples covered by C. The number of bits required to indicate that these examples are covered by C is $b_e(C) = \log_2(|E|) + \log_2(\binom{|E|}{p})$ [57]. The number of bits $b_c(C)$ required to encode clause C is equal to the sum of the number of bits required to encode each literal in C, reduced by $\log_2(m!)$, where m is the number of literals in C. The number of bits required to encode a literal is $1 + \log_2(|\mathcal{R}|) + \log_2(n)$ [57]. A clause C is in hypothesis space $\mathcal{L}$ if $b_c(C) \leq b_e(C)$.

Let relation $R_1(A, B, C)$ be in $\mathcal{R}$ and $\tau : \mathcal{R} \rightarrow \mathcal{S}$ decompose R_1 to $S_1(A, B)$ and $S_2(B, C)$. Let $T(A)$ be the target relation. Consider hypothesis $h_{\mathcal{R}}$: $T(x) \leftarrow R_1(x, y, z)$ over schema $\mathcal{R}$, whose mapped hypothesis is $h_{\mathcal{S}} = \delta_\tau(h_{\mathcal{R}}) = T(x) \leftarrow S_1(x, y), S_2(y, z)$. Then, $b_c(h_{\mathcal{R}}) = 1 + \log_2(1) + \log_2(3) - \log_2(1!)$ and $b_c(h_{\mathcal{S}}) = (1 + \log_2(2) + \log_2(3)) + (1 + \log_2(2) + \log_2(3)) - \log_2(2!)$.

Let $|E| = 5$ and $h_{\mathcal{R}}$ cover $p = 2$ examples. Because $h_{\mathcal{R}} \equiv h_{\mathcal{S}}$, then $b_e(h_{\mathcal{R}}) = b_e(h_{\mathcal{S}})$. Let $\mathcal{L}_{\mathcal{R}}^{FOIL}$ and $\mathcal{L}_{\mathcal{S}}^{FOIL}$ be the hypothesis spaces over $\mathcal{R}$ and $\mathcal{S}$, respectively. Hypothesis $h_{\mathcal{R}}$ is in $\mathcal{L}_{\mathcal{R}}^{FOIL}$ because $b_c(h_{\mathcal{R}}) \leq b_e(h_{\mathcal{R}})$, but $h_{\mathcal{S}}$ is not in $\mathcal{L}_{\mathcal{S}}^{FOIL}$ because $b_c(h_{\mathcal{S}}) > b_e(h_{\mathcal{S}})$. Therefore, the hypothesis spaces over schemas $\mathcal{R}$ and $\mathcal{S}$ are not equivalent. □

One may want to fix the problem of schema dependence in Progol by choosing different values for the maximum lengths over the original and transformed schemas. The following theorem states that it is not possible to achieve equivalent hypothesis spaces by restricting the maximum length of clauses no matter what values are used over the original and transformed schemas.

Theorem 3.4.2. *Progol is not hypothesis-invariant.*

Proof. Let relations $R_1(A, B, C)$ and $R_2(D, B, E)$ be in $\mathcal{R}$ and $\tau : \mathcal{R} \rightarrow \mathcal{S}$ decompose R_1 to $S_1(A, B)$ and $S_2(B, C)$ and R_2 to $S_3(D, B)$ and $S_4(B, E)$. Let l be the maximum clause length and $\gamma = \langle l \rangle$ be the parameter setting for Progol. Without loss of generality we set the value of l to 2. Let $T(x, y)$ be the target relation. Consider hypothesis $h_{\mathcal{R}}$: $T(x, y) \leftarrow R_1(x, z, w), R_2(y, z, v)$ over schema $\mathcal{R}$ whose mapped hypothesis $\delta_\tau(h_{\mathcal{R}})$ is: $T(x, y) \leftarrow S_1(x, z)$, $S_2(z, w)$, $S_3(y, z), S_4(z, v)$. Hypothesis $h_{\mathcal{R}}$ is in the hypothesis language $\mathcal{L}_{\mathcal{R},\gamma}^{Progol}$ because it clause length is equal to 2. However, hypothesis $\delta_\tau(h_{\mathcal{R}})$ is not in the hypothesis language $\mathcal{L}_{\mathcal{S},\gamma}^{Progol}$ because its clause length exceeds 2. Therefore, hypothesis spaces $\mathcal{L}_{\mathcal{R},\gamma}^{Progol}$ and $\mathcal{L}_{\mathcal{S},\gamma}^{Progol}$ are not equivalent. To achieve hypothesis equivalence, one may change the

parameter setting for $\mathcal{S}$ to γ' with $l = 4$ so that the hypothesis $\delta_{\mathcal{T}}(h_{\mathcal{R}})$ becomes a member of $\mathcal{L}^{Progol}_{\mathcal{S},\gamma'}$. This modification also brings the new hypothesis $T(x,y) \leftarrow S_1(x,z)$, $S_1(x,w)$, $S_1(x,t), S_1(x,y)$ to $\mathcal{L}^{Progol}_{\mathcal{S},\gamma'}$. The equivalent hypothesis to this new hypothesis over $\mathcal{R}$ is $T(x,y) \leftarrow R_1(x,z,v_1)$,$R_1(x,w,v_2)$, $R_1(x,t,v_3)$, $R_1(x,y,v_4)$ where v_i, $1 \leq i \leq 4$, is a fresh variable. Because this hypothesis over $\mathcal{R}$ is minimal, one has to also change l over $\mathcal{R}$ to 4 to achieve equivalent hypothesis spaces over $\mathcal{R}$ and $\mathcal{S}$. Hence, we have to alternate between the parameter settings over $\mathcal{R}$ and $\mathcal{S}$ without any stopping condition. Thus, there are not any fixed parameter settings that ensure the hypothesis equivalence over $\mathcal{R}$ and $\mathcal{S}$. □

3.5 Bottom-up Algorithms

Bottom-up algorithms also follow the covering approach shown in Algorithm 1. However, their *LearnClause* function searches the hypothesis space from specific to general hypotheses. Given a positive example, bottom-up algorithms build the most specific clause in the hypothesis space, called *bottom-clause*, that covers the example, relative to the database instance [48, 50]. Then, they generalize the bottom-clause to find a definition that covers as many positive and as few negative examples as possible. There are multiple bottom-up algorithms whose differences lie mainly in their generalization operator [7, 48, 50]. We consider two algorithms that are representative of the family of bottom-up algorithms: Golem [48] and ProGolem [50].

3.5.1 Bottom-clause Construction

Let I be a database instance over schema $\mathcal{R}$. The bottom-clause associated with positive example e, relative to I, denoted by $\perp_{e,I}$, is the most specific clause over $\mathcal{R}$ that covers e, relative to I [47]. The bottom-clause construction algorithm consists of two phases. First, it finds all the information in I relevant to e. Then, given the information relevant to e, it creates the bottom-clause $\perp_{e,I}$. 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 search operations, starting from e. Algorithm 2 depicts the bottom-clause construction algorithm. The algorithm maintains a set M that contains all seen constants. Let $e = T(a_1, \ldots, a_n)$ be a training example. First, the algorithm

adds $a_1, \ldots, a_n$ to M. These constants are values that appear in tuples in I. Then, it searches all tuples in I that contain at least one constant in M and adds them to I_e. For each new tuple in I_e, the algorithm extracts new constants and adds them to M. The algorithm repeats this process for a fixed number of iterations d. To create the bottom-clause C_e from I_e, the algorithm 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 $t \in I_e$, it creates a literal and adds it to the body of the clause, replacing each constant in t with its assigned variable. The algorithm may *not* replace some constants with variables if specified by a given language bias, as explained in Chapter 4.

Algorithm 2: Bottom-clause construction algorithm.

```
Input  : example e, database instance I, # of iterations d
Output: bottom-clause C_e
I_e = {}
M = {} // M stores known constants
add constants in e to M
for i = 1 to d do
    foreach relation R ∈ I do
        foreach attribute A in R do
            I_R = σ_{A∈M}(R)
            foreach tuple t ∈ I_R do
                add t to I_e and constants in t to M
⊥_{e,I} = create clause from e and I_e
return ⊥_{e,I}
```

Example 3.5.1. *Given example e=advisedBy(s1, p1) and the database instance in Table 3.2, the bottom-clause associated with e is:*

$$\begin{aligned} advisedBy(x,y) \leftarrow & student(x, \text{`}post_generals\text{'}, \text{`}5\text{'}), professor(y, \text{`}faculty\text{'}), publication(v,x), \\ & publication(v,y), ta(u,x,\text{`}spring_2010\text{'}), taughtBy(u,y,\text{`}spring_2010\text{'}) \\ & courseLevel(u,\text{`}graduate\text{'}). \end{aligned}$$

Algorithm 2 may generate very long clauses after multiple iterations. Therefore, the number of iterations must be restricted by the parameter d. The number of iter-

student(s1, post_generals, 5)	professor(p1, faculty)	courseLevel(c1, graduate)
student(s2, post_generals, 7)	professor(p2, faculty)	ta(c1, s1, spring_2010)
publication(t1, s1)	publication(t1, p1)	taughtBy(c1, p1, spring_2010)
publication(t2, s2)	publication(t2, p2)	

Table 3.2: Example database over the 4NF schema of the UW-CSE dataset.

ations limits the maximum *depth* of the bottom-clause [47]. The depth of a variable x, denoted by $depth(x)$, is 0 if it appears in the head of the clause, otherwise it is $min_{v \in U_x}(depth(v)) + 1$, where U_x are the variables of literals in the body of the clause containing x. The depth of a literal is the maximum depth of the variables appearing in the literal. The depth of a clause is the maximum depth of the literals appearing in the clause. The algorithm creates literals of depth at most i in iteration i.

Example 3.5.2. *The following clause, defined over the Original schema of the UW-CSE database in Table 3.1, has depth 1:*

$$taLevel(x, y) \leftarrow ta(c, x, t), courseLevel(c, y).$$

On the other hand, the following clause, defined over the same schema, has depth 2:

$$commonLevel(x, y) \leftarrow ta(c1, x, t1), ta(c2, y, t2), courseLevel(c1, l), courseLevel(c2, l).$$

Bottom-clauses determine the hypothesis space of a bottom-up algorithm: longer bottom-clauses allow the algorithm to explore a larger number of definitions. To be schema independent, bottom-up algorithms must get equivalent bottom-clauses associated with the same example, relative to equivalent instances of the original and transformed schemas. Otherwise, these algorithms will not be hypothesis-invariant. Using the parameter d, which restricts the maximum depth of bottom-clauses, does not result in such equivalent bottom-clauses because the original and transformed schemas need different depths to create equivalent bottom-clauses. Therefore, the bottom-clause construction algorithm is *not* schema independent.

Example 3.5.3. *Let us compose and replace relations courseLevel(crs, level) and ta(crs, stud, term) in the Original schema of the UW-CSE database with relation courseLevelTa(crs, level, stud, term). The definition for target relation commonLevel from Example 3.5.2*

over this schema is the following:

$$commonLevel(x, y) \leftarrow courseLevelTa(c1, l, x, t1), courseLevelTa(c2, l, y, t2).$$

This definition has depth 1. If we set the maximum depth to 1, this definition is in the hypothesis language. However, the definition in Example 3.5.2 defined over the Original schema is not in the hypothesis language because it contains variables that have depth 2.

Lemma 3.5.4. *The bottom-clause construction algorithm is not schema independent.*

Proof. Proven by contradiction with Example 3.5.3. □

3.5.2 Golem

In this section, we consider a bottom-up learning algorithm called Golem [48]. Golem, like other learning algorithms, follows a covering approach, as the one shown in Algorithm 1. Golem's *LearnClause* function follows a bottom-up approach, which is based on the *least general generalization* (*lgg*) operator. Given clauses C_1 and C_2, the *lgg* of C_1 and C_2 is the clause C that is more general than C_1 and C_2, but the least general such clause. The notion of generality is given by θ-subsumption (defined in Section 2.1). Therefore, clause C is more general than C_1 if and only if C θ-subsumes C_1 (and similarly for C_2). This notion of generality gives a computable generality relation.

The *lgg* of two constants a_1 and a_2, such that $a_1 \neq a_2$, is $lgg(a_1, a_2) = v$, where v is a new variable. The *lgg* of two constants that are equal, i.e., constant a, is $lgg(a, a) = a$. The *lgg* of a constant a and a variable v is $lgg(a, v) = v$. The *lgg* of two variables v_1 and v_2, such that $v_1 \neq v_2$, is $lgg(v_1, v_2) = v$, where v is a new variable. The *lgg* of two variables that are equal, i.e., variable v, is $lgg(v, v) = v$. Two atoms are *compatible* if they have the same relation name and the same arity. The *lgg* of two compatible atoms $R(u_1, \cdots, u_k)$ and $R(u'_1, \cdots, u'_k)$, where u_i and u'_i are variables or constants, is $lgg(R(u_1, \cdots, u_k), R(u'_1, \cdots, u'_k)) = R(lgg(u_1, u'_1), \cdots, lgg(u_k, u'_k))$. The *lgg* of two atoms that are not compatible is undefined. Two literals are *compatible* if their atoms are compatible and they have the same polarity. The *lgg* of two compatible literals is the *lgg* of its atoms. The *lgg* of two literals that are not compatible is undefined. The *lgg* of two clauses C_1 and C_2 is the set of pairwise *lgg* operations of compatible literals in C_1 and C_2. When computing the *lgg* of two clauses C_1 and C_2, the application

of the *lgg* operator for a constant or variable v_1 in C_1 and a constant or variable v_2 in C_2, i.e., $lgg(v_1, v_2)$ always returns the same variable v. The *lgg* of two clauses is unique. The *lgg* of a set of clauses $\{C_1, ..., C_{n-1}\}$ is defined via pairwise operations: $lgg(\{C_1, ..., C_n\}) = lgg(lgg(\{C_1, ..., C_{n-1}\}), C_n)$. The order of pairwise applications of the *lgg* operation does not matter as the *lgg* operator is commutative and associative.

Example 3.5.5. *Consider the following clauses:*

$$\begin{aligned} C_1 = advisedBy(v_1, v_2) \leftarrow & student(v_1, \text{`}post_generals\text{'}, \text{`}5\text{'}), \\ & publication(v_3, v_1), publication(v_3, v_2) \\ & ta(v_4, v_1, \text{`}spring_2010\text{'}), taughtBy(v_4, v_2, \text{`}spring_2010\text{'}) \\ C_2 = advisedBy(v_{11}, v_{12}) \leftarrow & student(v_{11}, \text{`}post_generals\text{'}, \text{`}7\text{'}), \\ & publication(v_{13}, v_{11}), publication(v_{13}, v_{12}). \end{aligned}$$

Then, $lgg(C_1, C_2)$ *is the following clause:*

$$\begin{aligned} advisedBy(v_{21}, v_{22}) \leftarrow & student(v_{21}, \text{`}post_generals\text{'}, v_{23}), \\ & publication(v_{24}, v_{21}), publication(v_{24}, v_{25}), \\ & publication(v_{24}, v_{26}), publication(v_{24}, v_{22}). \end{aligned}$$

A *ground bottom-clause* is a bottom-clause that only contains constants. Given the ground bottom-clauses of two examples e_1 and e_2, the operator that computes the *lgg* of the two ground bottom-clauses is called the *relative least general generalization* (*rlgg*) of e_1 and e_2.

Given a database instance I and training examples E^+ and E^-, Golem's *LearnClause* function learns a clause that covers as many positive and as few negative examples as possible. Algorithm 3 sketches this function. Intuitively, the algorithm first randomly selects a subset E_S^+ of positive examples E^+. It then generates candidate clauses by computing the *rlgg* between every pair of examples in E_S^+. The algorithm considers only candidate clauses that satisfy some minimum condition, e.g., minimum precision of a clause. It then greedily includes new examples into the generalization to create new candidate clauses. This algorithm uses the function $Covers(C, E)$, which returns the examples in E covered by clause C. The algorithm stops when no improvement can be

made.

Algorithm 3: Golem's *LearnClause* algorithm.

Input : Database instance I, positive examples E^+, negative examples E^-, parameter K.
Output: A new clause C^*.
$E_S^+ = K$ randomly selected positive examples from E^+
$\mathbf{C} = \{C = lgg(\bot_{e,I}, \bot_{e',I}) \mid e, e' \in E_S^+, C$ satifies minimum condition$\}$
while $\mathbf{C}$ *is not empty* **do**
  $C^* = \text{argmax}_{C \in \mathbf{C}}\ Score(C, E_S^+, E^-)$
  $E_S^+ = E_S^+ - Covers(C^*, E_S^+)$
  $\mathbf{C} = \{C = lgg(C^*, \bot_{e,I}) \mid e \in E_S^+, C$ satifies minimum condition$\}$
return C^*

Theorem 3.5.6. *The rlgg operator is schema independent.*

Proof. Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a bijective transformation that performs compositions or decompositions from relations in schema $\mathcal{R} = (\mathbf{R}, \Sigma_{\mathcal{R}})$ to relations in schema $\mathcal{S} = (\mathbf{S}, \Sigma_{\mathcal{S}})$. Let I and J be instances of $\mathcal{R}$ and $\mathcal{S}$, respectively, such that $\tau(I) = J$. Let T be the target relation, and $e_1 = T(a_1, \cdots, a_l)$ and $e_2 = T(b_1, \cdots, b_l)$ be two positive examples. Let $(e_1 \leftarrow I'_1)$ and $(e_2 \leftarrow I'_2)$ be the ground bottom-clauses under schema $\mathcal{R}$ for e_1 and e_2, respectively, such that $I'_1, I'_2 \subseteq I$. Similarly, let $(e_1 \leftarrow J'_1)$ and $(e_2 \leftarrow J'_2)$ be the ground bottom-clauses under schema $\mathcal{S}$ for e_1 and e_2, respectively, such that $J'_1, J'_2 \subseteq J$.

We show that the *rlgg* of examples e_1 and e_2 is equivalent under schemas $\mathcal{R}$ and $\mathcal{S}$. That is

$$rlgg_{\mathcal{R}}(e_1, e_2) \equiv rlgg_{\mathcal{S}}(e_1, e_2)$$
$$lgg((e_1 \leftarrow I'_1), (e_2 \leftarrow I'_2)) \equiv lgg((e_1 \leftarrow J'_1), (e_2 \leftarrow J'_2))$$

The *lgg* of clauses $(e_1 \leftarrow I'_1)$ and $(e_2 \leftarrow I'_2)$ is the set of pairwise *lgg* operations of compatible ground literals in $(e_1 \leftarrow I'_1)$ and $(e_2 \leftarrow I'_2)$. We show that the *lgg* of compatible ground literals under schema $\mathcal{R}$ delivers equivalent results under schema $\mathcal{S}$.

Let $R \in \mathbf{R}$ be a relation in $\mathcal{R}$ such that $\tau(R) = S_1, \cdots, S_m$, $1 \leq m \leq |\mathbf{S}|$. Because of Corollary 4.3.2 by Atzeni et al. [8], we know that if τ is bijective, $\Sigma_{\mathcal{S}}$ contains inclusion dependencies between the join attributes of $S_1, \cdots, S_m$. Let $r_1 = R(a_1, \cdots, a_k)$ and

$r_2 = R(a'_1, \cdots, a'_k)$ be two ground atoms in I. Then, $\tau(r_1) = S_1(t_1), \cdots, S_m(t_m)$ and $\tau(r_2) = S_1(t'_1), \cdots, S_m(t'_m)$ are ground atoms in J, where t_i and t'_i, $1 \leq i \leq m$, are tuples. Then, the *lgg* of ground atoms r_1 and r_2 is defined as

$$lgg(r_1, r_2) = R(lgg(a_1, a'_1), \cdots, lgg(a_k, a'_k))$$

By applying transformation τ, $R(lgg(a_1, a'_1), \cdots, lgg(a_k, a'_k))$ is equivalent to

$$S_1(s_1), S_2(s_2), \cdots, S_m(s_m)$$

where s_j is a tuple that contains a subset of attributes in $\{lgg(a_1, a'_1), \cdots, lgg(a_k, a'_k)\}$ for $1 \leq j \leq m$. By definition of the *lgg* operator, we get

$$\begin{aligned} S_1(s_1), S_2(s_2), \cdots, S_m(s_m) =& lgg(S_1(t_1), S_1(t'_1)), \cdots, lgg(S_m(t_m), S_m(t'_m)) \\ =& lgg(\tau(r_1), \tau(r_2)) \end{aligned}$$

□

In Section 3.6.1 we show that the bottom-clause construction algorithm can be modified to be schema independent. Because the *rlgg* operator is also schema independent, Golem can achieve schema independence. However, the size of the clauses generated by the *rlgg* operator may grow exponentially with the number of positive training examples, as we explain next. Therefore, Golem is not an efficient learning algorithm. Let C_1 and C_2 be two clauses. The *lgg* of C_1 and C_2 is the set of pairwise *lgg* operations of compatible literals in C_1 and C_2. If all the literals in the bodies of C_1 and C_2 are compatible, then the clause generated by $lgg(C_1, C_2)$ contains a literal for every pair of literals in C_1 and C_2. Therefore, the size of $lgg(C_1, C_2)$ is $|C_1| \times |C_2|$, where $|C_1|$ and $|C_2|$ are the sizes of C_1 and C_2, respectively. As seen in Algorithm 3, given a pair of positive examples, Golem applies the *lgg* operator to obtain a new clause C. It then computes the *lgg* between C and the ground bottom-clause of a new positive example. Golem repeats this process until all examples are covered. Let n be the number of positive examples to generalize and m be the maximum length of a ground bottom-clause. Then, the length of the clause generated by Golem is bounded by $O(|C_1| \times \cdots \times |C_n|) = O(m \times \cdots \times m) = O(m^n)$, i.e., it grows exponentially with the number of positive examples covered. Therefore, an

algorithm that uses the *rlgg* operator, such as Golem, cannot learn efficiently without making assumptions that do not hold over most real-world databases [50].

3.5.3 ProGolem

ProGolem is a bottom-up algorithm that can run efficiently over small or medium databases without making generally unrealistic assumptions [50]. To explore the hypothesis space and generalize clauses efficiently, ProGolem assumes that clauses are ordered. An *ordered clause* is a clause where the order and duplication of literals matter. If clause C is considered an ordered clause, then it is denoted as $\overrightarrow{C}$. For instance, clauses $\overrightarrow{C} = T(x) \leftarrow P(x), Q(x)$, $\overrightarrow{D} = T(x) \leftarrow Q(x), P(x)$, and $\overrightarrow{E} = T(x) \leftarrow P(x), P(x), Q(x)$ are all different.

ProGolem uses the *asymmetric relative minimal generalization* (*armg*) operator to generalize clauses. ProGolem's *LearnClause* function first generates the bottom-clause associated with some positive example. Then, it performs a beam search to select the best clause generated after multiple applications of the *armg* operator. More formally, given clause $\overrightarrow{C}$, ProGolem randomly picks a subset E_S^+ of positive examples to generalize $\overrightarrow{C}$. For each example e' in E_S^+, ProGolem uses the *armg* operator to generate a candidate clause $\overrightarrow{C'}$, which is more general than $\overrightarrow{C}$ and covers e'. It then selects the highest scoring candidate clauses to keep in the beam and iterates until the clauses cannot be improved. The beam search requires an evaluation function to score clauses. One may select an evaluation function that is agnostic to the schema used, such as coverage, which is the number of positive examples minus the number of negative examples covered by the clause.

Algorithm 4: ProGolem's *ARMG* algorithm.

Input : Bottom-clause $\bot_{e,I_{\mathcal{R}}}$, positive example e'.
Output: An *ARMG* of $\bot_{e,I_{\mathcal{R}}}$ that covers e'.
$\overrightarrow{C}$ is an ordered version of $\bot_{e,I_{\mathcal{R}}} = T \leftarrow L_1, \cdots, L_n$
while *there is a blocking atom* L_i *in the body of* $\overrightarrow{C}$ **do**
    Remove L_i from $\overrightarrow{C}$
    Remove atoms from $\overrightarrow{C}$ which are not head-connected
Return $\overrightarrow{C}$

We now explain the *armg* operator in detail. Let $\bot_{e,I_{\mathcal{R}}}$ be the bottom-clause associated with example e, relative to $I_{\mathcal{R}}$. Let $\overrightarrow{C} = T \leftarrow L_1, \cdots, L_n$ be an ordered version of $\bot_{e,I_{\mathcal{R}}}$. Let e' be another example. L_i is a *blocking atom* if and only if i is the least value such that for all substitutions θ where $e' = T\theta$, the clause $\overrightarrow{C'}\theta = (T \leftarrow L_1, \cdots, L_i)\theta$ does not cover e', relative to $I_{\mathcal{R}}$ [50]. Algorithm 4 shows the *ARMG* algorithm, which implements the *armg* operator. Given the bottom-clause $\bot_{e,I_{\mathcal{R}}}$ and a positive example e', *armg* drops all blocking atoms from the body of $\bot_{e,I_{\mathcal{R}}}$ until e' is covered. After removing a blocking atom, some literals in the body may not have any variable in common with the other literals in the body and head of the clause, i.e., they are not *head-connected*. *ARMG* also drops these literals.

For ProGolem to be schema independent, the *armg* operator must return equivalent clauses given equivalent input clauses over original and transformed databases.

Example 3.5.7. *Consider the following equivalent definitions for target relation hardWorking over the Original and 4NF UW-CSE schema in Table 3.1, respectively:*

$$hardWorking(x) \leftarrow student(x), inPhase(x, \text{‘}prelim\text{’}), yearsInProgram(x, \text{‘}3\text{’}), publication(z, x)$$
$$hardWorking(x) \leftarrow student(x, \text{‘}prelim\text{’}, \text{‘}3\text{’}), publication(z, x).$$

Assume that we use armg to generalize these clauses to cover example e'. Let e' satisfy literal $student(x)$ but not satisfy literal $inPhase(x, \text{‘}prelim\text{’})$. The armg operator keeps literal $student(x)$ in the first clause, but it eliminates $student(x, \text{‘}prelim\text{’}, \text{‘}3\text{’})$ from the second clause. Hence, it delivers non-equivalent generalizations.

As stated in Lemma 3.5.4, the bottom-clause construction algorithm is not schema independent. Thus, neither the bottom-clause construction nor the generalization phases in ProGolem are schema independent.

Theorem 3.5.8. *ProGolem is not schema independent.*

3.6 Castor

This section presents *Castor*, a bottom-up relational learning algorithm. Castor uses the covering approach presented in Algorithm 1. It follows the same search strategy as ProGolem, but integrates inclusion dependencies (INDs) into the bottom-clause construction

and generalization algorithms to achieve schema independence. INDs are normally stored in the schema of the database. If they are not available in the schema, one can extract them from the database content, as seen in Chapter 4. If we apply the INDs in schema $\mathcal{R}$ to Horn clause $h_{\mathcal{R}}$ over $\mathcal{R}$, we get an equivalent Horn clause that has a similar syntactic structure to its equivalent Horn clause in any decomposition of $\mathcal{R}$ [2]. For example, consider schema $\mathcal{R}$:$\{R_1(A, B), R_2(A, C)\}$ with the IND $R_1[A] = R_2[A]$ and the clause $h_{\mathcal{R}}$: $T(x) \leftarrow R_1(x, y)$. Because each value in $R_1[A]$ also appears in $R_2[A]$, we can rewrite $h_{\mathcal{R}}$ as $g_{\mathcal{R}} : T(x) \leftarrow R_1(x, y), R_2(x, z)$. Now, consider a composition of $\mathcal{R}$, $\mathcal{S}$:$\{S_1(A, B, C)\}$. The clause $h_{\mathcal{S}} : T(x) \leftarrow S_1(x, y, z)$ over $\mathcal{S}$ is equivalent to both $h_{\mathcal{R}}$ and $g_{\mathcal{R}}$. Clauses $g_{\mathcal{R}}$ and $h_{\mathcal{S}}$ have also similar syntactic structures: there is a bijection between the distinct variables in $g_{\mathcal{R}}$ and $h_{\mathcal{S}}$. However, such bijection does not exist between $h_{\mathcal{R}}$ and $h_{\mathcal{S}}$. As learning algorithms modify the syntactic structure of clauses to learn a target definition and $h_{\mathcal{R}}$ and $h_{\mathcal{S}}$ have different syntactic structures, these algorithms may modify them differently and generate non-equivalent clauses. For instance, assume that an algorithm renames variable z to x in $h_{\mathcal{S}}$ to generate clause $h^{'}_{\mathcal{S}} : T(x) \leftarrow S_1(x, y, x)$. This algorithm cannot apply a similar change to $h_{\mathcal{R}}$ as $h_{\mathcal{R}}$ does not have any corresponding variable to z. But, the algorithm can apply the same modification to $g_{\mathcal{R}}$ and generate an equivalent Horn clause to $h^{'}_{\mathcal{S}}$. Moreover, as INDs generally reflect important relationships, they can be used by the algorithm for improving the effectiveness of the learned definitions.

Castor's *LearnClause* function is shown in Algorithm 5. It first generates the bottom-clause associated with some positive example using the modified bottom-clause construction algorithm presented in Section 3.6.1. It minimizes the bottom-clause using the procedure explained in Section 3.6.5. Then, it performs a beam search to select the best candidate after multiple applications of the modified *ARMG* algorithm, explained in Section 3.6.2.1. Finally, it reduces the best candidate using the algorithm explained in Section 3.6.2.2.

3.6.1 Castor Bottom-Clause Construction

Castor selects a positive example and constructs its bottom-clause by following the normal procedure of bottom-clause construction: at each iteration, it selects a relation and adds one or more literals of that relation to the bottom-clause. Let relation symbol R in the schema $\mathcal{R}$ be decomposed to relation symbols $S_1 \dots S_n$ in the transformed schema

Algorithm 5: Castor's *LearnClause* algorithm.

Input : Database instance I, positive examples E^+, negative examples E^-, parameters K and N.

Output: A new clause C.

$\overrightarrow{C} = \mathit{Castor_BottomClause}(\text{first example in } E^+)$
$\overrightarrow{C} = \mathit{Minimize}(\overrightarrow{C})$
$BC = \{\overrightarrow{C}\}$
repeat
  $\mathit{BestScore}$ = highest score of candidates in BC
  $E_S^+ = K$ randomly selected positive examples from E^+
  $NC = \{\}$
  foreach *clause* $C \in BC$ **do**
    foreach $e' \in E_S^+$ **do**
      $C' = \mathit{Castor_ARMG}(C, e')$
      if $\mathit{Score}(C') > \mathit{BestScore}$ **then**
        $NC = NC \cup C'$
  BC = highest scoring N candidates from NC
until $NC = \{\}$
C' = highest scoring candidate in BC
return $\mathit{Castor_Reduce}(C', I, E^-)$

$\mathcal{S}$. If the bottom-clause construction algorithm considers tuple r in an instance of R, I_R, it must also examine tuples $s_1, \ldots, s_n$ in instances $I_{S_1}, \ldots, I_{S_n}$, respectively, such that $\bowtie_{i=1}^{n} [s_i] = r$, to ensure that the produced bottom-clauses over both schemas are equivalent. After the bottom-clause construction algorithm replaces the constants with variables in these bottom-clauses, it generates equivalent bottom-clauses over $\mathcal{R}$ and $\mathcal{S}$. Hence, if Castor examines tuple $s_j \in I_{S_j}$, it should find tuples $s_i \in I_{S_i}$ whose natural join with s_j creates tuple r. One approach is to find all relations S_i that have some common attributes with S_j as they have some tuples that join with s_i and produce r. However, designers may rename the attributes on which $S_1 \ldots S_n$ join. For instance, relations *student*, *inPhase*, and *yearsInProgram* in the original schema join over attribute *stud* to create relation *student* in the 4NF schema in Table 3.1. The database designer may rename attribute *stud* to *name* in relation *student*. Hence, this approach is not robust against attribute renaming. According to Definition 3.3.1, there are INDs with equality between the join attributes of relation symbols $S_1 \ldots S_n$. We use INDs with equality between the attributes in schema $\mathcal{S}$ to find tuples s_i. To simplify our notations, we assume that the join between relations in $\mathcal{S}$ is still natural join. Our results extend for composition joins that are equi-join.

Definition 3.6.1. *The inclusion class* $\mathbf{N}$ *in schema* $\mathcal{S}$ *is the maximal set of relation symbols in* $\mathcal{S}$ *such that for each* $S_i, S_j \in \mathbf{N}$, $i \neq j$, *there is a sequence of INDs* $S_k[X_k] = S'_k[X_k]$, $i \leq k \leq j$, *in* $\mathcal{S}$ *such that*

1. $X_k = sort(S_k) \cap sort(S'_k)$.
2. $S_{k+1} = S'_k$ *for* $i \leq k \leq j-1$.

Castor first constructs the inclusion classes in the input schema $\mathcal{S}$. Assume that the algorithm generates a bottom-clause relative to an instance of schema $\mathcal{S}$. Also, assume that the algorithm has just selected relation I_{S_i} and added literal L_i to the bottom-clause based on some tuple s_i of I_{S_i}. Let S_i be a member of inclusion class $\mathbf{N}$ in $\mathcal{S}$. For each constraint $S_j[X] = S_i[X]$ between the members of $\mathbf{N}$, Castor selects all tuples s_j of relation I_{S_j}, $i \neq j$ such that $\pi_X(s_j) = \pi_X(s_i)$. It applies the same process for s_j until it exhausts the INDs between the members of $\mathbf{N}$. As the join between $S_1 \ldots S_n$ is pairwise consistent, this method efficiently finds the all tuples $s_1, \ldots, s_n$ that participate in the join and none of them is a dangling tuple with regard to the full join. Otherwise, Castor must check the join condition for each pair of tuples.

Example 3.6.2. *Consider an instance of the original UW-CSE schema in Table 3.1 with tuples s_1: student(Abe), s_2: inPhase(Abe, prelim) and s_3: yearsInProgram(Abe, 3). Assume that Castor is building a bottom-clause for example hardWorking(Abe). Given INDs student[stud] = inPhase[stud] and student[stud] = yearsInProgram[stud] hold in this schema, student, inPhase, and yearsInProgram constitute an inclusion class. Let Castor select tuple s_1 during the bottom-clause construction and add it to the bottom-clause. The bottom-clause, before replacing constants with variables, is:*

$$hardWorking(\text{`}Abe\text{'}) \leftarrow student(\text{`}Abe\text{'}).$$

As $\pi_{stud}(s_1) = \pi_{stud}(s_2)$ and $\pi_{stud}(s_1) = \pi_{stud}(s_3)$, Castor adds tuples s_2 and s_3 to the bottom-clause:

$$hardWorking(\text{`}Abe\text{'}) \leftarrow student(\text{`}Abe\text{'}), inPhase(\text{`}Abe\text{'}, \text{`}prelim\text{'}), yearsInProgram(\text{`}Abe\text{'}, \text{`}3\text{'}).$$

The INDs between relations in a inclusion class may form a cycle.

Definition 3.6.3. *A set of INDs with equality λ over schema $\mathcal{S}$ is* cyclic *if there is a sequence $S_i[X_i] = S'_i[Y_i]$, $1 \leq i \leq n$, in λ such that*

1. *$S_{i+1} = S'_i$ for $1 \leq i \leq n-1$ and $S_1 = S'_n$.*
2. *There is an i where $Y_i \neq X_{i+1}$.*

If the INDs induced by the inclusion class $\mathbf{N}$ are cyclic, Castor may have to examine a lot more tuples than the case where the INDs of $\mathbf{N}$ are not cyclic. For example, consider schema $\mathcal{S}_1$ with relations $S_1(A, B)$, $S_2(B, C)$, and $S_3(C, A)$. The set of INDs $S_1[B] = S_2[B]$, $S_2[C] = S_3[C]$, and $S_3[A] = S_1[A]$ is cyclic. Consider tuples s_1, s_2, and s_3 such that $\pi_B(s_1) = \pi_B(s_2)$ and $\pi_C(s_2) = \pi_C(s_3)$. We may not have $\pi_A(s_3) = \pi_A(s_1)$. Hence, Castor has to scan many tuples in S_3 to find a tuple s'_3 that satisfies both $\pi_C(s_2) = \pi_C(s'_3)$ and $\pi_A(s'_3) = \pi_A(s_1)$. The following proposition shows that if the composition join in Definition 3.3.1 is acyclic, the INDs with equality in the decomposed schema are not cyclic. Thus, Castor does not face the aforementioned issue.

Proposition 3.6.4. *Give schema $\mathcal{R}$ with a single relation symbol R and its decomposition $\mathcal{S}$ with relation symbols $S_1, \ldots, S_n$, if the join $\bowtie_{j=1}^{n} [S_1, \ldots, S_n]$ is acyclic, the INDs with equality λ in Definition 3.3.1 are not cyclic.*

Proof. Because the join is acyclic, there is a join tree for it whose nodes are S_i, $1 \leq i \leq n$ such that (i) every edge (S_i, S_j) is labeled by the set of attributes $sort(S_i) \cap sort(S_j)$ and (ii) for every pair S_i, S_j of distinct nodes, for each attribute $A \in sort(S_i) \cap sort(S_j)$, each edge along the unique path between S_i and S_j includes label A. As the IND with equalities λ are defined over the common attributes of S_i and S_j, λ are acyclic. □

Given $S_i, S_j \in \mathbf{N}$, too many tuples from a relation I_{S_j} may join with the current tuple $s_i \in I_{S_i}$, which may result in an extremely large bottom-clause. One may limit the maximum number of tuples that can join with the current tuple to a reasonably large value. We use the value of 10 in our reported experiments. After finding the join tuples, for each tuple s_j, Castor creates a ground literal L_j. If a constant in L_j has been already seen, the algorithm replaces it in L_j with the variable that was assigned to that constant. Otherwise, it assigns a fresh new variable for that constant in L_j. Finally, the algorithm adds L_j to the bottom-clause. Because inclusion classes are maximal, each relation symbol belongs to at most one inclusion class. After exhausting all INDs with equality between the members of $\mathbf{N}$, Castor returns to the typical procedure of bottom-clause construction. Castor may scan more relations than other bottom-clause construction algorithms to find tuples that satisfy the INDs at the end of each iteration. But, a schema usually has a relatively small number of INDs. We show in Sections 3.6.5 and 3.8 that Castor's bottom-clause construction algorithm runs faster than other algorithms.

As explained in Section 3.5.1, the bottom-clauses may get too large. We propose a modification of the original bottom-clause construction algorithm so that the stopping condition is based on the maximum number of distinct variables in a bottom-clause. At the end of each iteration, Castor checks how many distinct variables are in the bottom-clause. If this number is less than an input parameter, Castor continues to the next iteration; otherwise, it stops. Intuitively, since the number of distinct variables in equivalent Horn clauses over composition and decomposition are equal, this condition helps Castor to return equivalent bottom-clauses over composition and decomposition. The following Lemma states that Castor's bottom-clause construction algorithm is schema independent.

Lemma 3.6.5. *Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a decomposition and $\tau^{-1} : \mathcal{S} \rightarrow \mathcal{R}$ be a composition, I be an instance of $\mathcal{R}$, and $\bot_{e,I}$ and $\bot_{e,\tau(I)}$ are bottom-clauses generated by Castor for example e relative to I and $\tau(I)$, respectively. We have $\bot_{e,I} \equiv \bot_{e,\tau(I)}$.*

Proof. Assume that τ decomposes I_R to relations $J_{S_1}, \ldots, J_{S_m}$. Let the constants in e appear in a subset of relation I_R denoted as I_R^e. Thus, the constants in e must also appear in at least a subset of one relation in $\tau(I)_{S_1}, \ldots, \tau(I)_{S_m}$, shown as $\tau(I)^e_{S_i}$, $1 \leq i \leq m$. The algorithm examines all tuples in I_R^e and $\tau(I)^e_{S_i}$ at the same iteration. Let L be the set of literals that the algorithm adds to $\bot_{e,I}$ based on tuples in I_R^e. By applying INDs, the algorithm considers all tuples s_j in $J_{S_1}, \ldots, J_{S_m}$ such that $\bowtie_{j=1}^{n} [s_j] \equiv r$ for every $r \in I_R^e$. Hence, it will create equivalent clauses at the current iteration. In the following iterations, as the algorithm selects tuples in I and $\tau(I)$ using the same set of constants, it adds equivalent literals to the clauses over I and $\tau(I)$. Because the algorithm uses a one-to-one mapping from variables to constants, the clauses over I and $\tau(I)$ will be equivalent when the algorithm stops.

Now assume that J is an instance of schema $\mathcal{S}$ and τ^{-1} composes relations $J_{S_1}, \ldots, J_{S_m}$, where $J_{S_i} \subseteq J$, to relation I_R. Let the constants in e appear in at least one relation J_{S_i} denoted as $J^e_{S_i}$. By applying INDs to each tuple in $J^e_{S_i}$, the algorithm finds all tuples s_j in $J_{S_1}, \ldots, J_{S_m}$ such that $\bowtie_{j=1}^{n} [s_j] \equiv r \in \tau^{-1}(J_{S_1} \ldots J_{S_m}) \subseteq I_R$. Thus, the constants in e must also appear in a subset of relation I_R. Hence, it will create equivalent clauses at the current iteration. In the following iterations, as the algorithm selects tuples in J and $\tau^{-1}(J)$ using the same set of constants, it adds equivalent literals to the clauses over J and $\tau^{-1}(J)$. Because the algorithm uses a one-to-one mapping from variables to constants, the clauses over J and $\tau^{-1}(J)$ will be equivalent when the algorithm stops. $\square$

3.6.2 Castor Generalization

3.6.2.1 *ARMG* Algorithm

Castor modifies Algorithm 4 to compute equivalent clauses generated by the *armg* operator over composition and decomposition. Before we explain the Castor generalization algorithm, we define some concepts. Consider clause $\overrightarrow{C}$ and literal $R(u)$ in $\overrightarrow{C}$, where u is a tuple. If u contains both variables and constants, we call it a *free tuple*. Let X be a set of attributes. The projection $\pi_X(u)$ of free tuple u returns the variables or constants that appear in attributes X in u. The join $u_1 \bowtie_{X=Y} u_2$ between free tuples u_1 and u_2 returns a join tuple if $\pi_X(u_1) = \pi_Y(u_2)$, where $\pi_X(u_1)$ and $\pi_Y(u_2)$ may contain constants or variables. We assume that all join operations are natural joins, i.e.,

$X = Y$. Our results extend for the case that joins are equi-joins. A *canonical database instance* of clause $\overrightarrow{C}$, shown as $I^{\overrightarrow{C}}$, is the database instance whose tuples are the free tuples in $\overrightarrow{C}$ [2]. In other words, relation I_R in $I^{\overrightarrow{C}}$ has free tuple u if literal $R(u)$ is in $\overrightarrow{C}$. Consider an IND $R_1[X] = R_2[X]$. The canonical database instance $I^{\overrightarrow{C}}$ satisfies the IND $R_1[X] = R_2[X]$ if for any literal $R_2(u_1)$ (or $R_2(u_2)$) in $\overrightarrow{C}$, clause $\overrightarrow{C}$ contains a literal $R_2(u_2)$ (or $R_1(u_1)$) such that $\pi_X(u_1) = \pi_X(u_2)$, where $\pi_X(u_1)$ and $\pi_X(u_2)$ may contain constants or variables. Otherwise, $I^{\overrightarrow{C}}$ violates the IND. If $I^{\overrightarrow{C}}$ satisfies the IND $R_1[X] = R_2[X]$ for literals $R_1(u_1)$ and $R_2(u_2)$, we say that free tuples u_1 and u_2 *satisfy* the IND.

In each iteration Castor's *ARMG* algorithm, Castor ensures that the canonical database instance of clause $\overrightarrow{C}$ always satisfies the INDs of the schema. Assume the algorithm is applied on instance $I_{\mathcal{R}}$ of schema $\mathcal{R} = (\mathbf{R}, \Sigma)$. Immediately after removing a blocking atom L_i from clause $\overrightarrow{C}$ in Algorithm 4, Castor examines all remaining literals in $\overrightarrow{C}$ and finds the ones whose relation symbols participate in an IND with equality in Σ. More precisely, let $R_1(u_1)$ be a literal and $\lambda_{R_1} \subseteq \Sigma$ be the set of INDs with equality in which R_1 participates. For each IND $R_1[X] = R_2[X]$ in λ_{R_1}, if there is *not* a literal with relation symbol R_2 in $\overrightarrow{C}$, Castor eliminates literal $R_1(u_1)$ from $\overrightarrow{C}$. Otherwise, assume that $\overrightarrow{C}$ contains literal $R_2(u_2)$. If for all literals $R_2(u_2)$, we have $\pi_X(u_1) \neq \pi_X(u_2)$, Castor removes literal $R_1(u_1)$. Castor checks these conditions for every literal in $\overrightarrow{C}$ and all its corresponding INDs. Castor increases the time complexity of Algorithm 4 by a factor of $O(|C_{max}|^2|\lambda|)$, where the $|C_{max}|$ is the size of the largest candidate clause and $|\lambda|$ is the number of INDs with equality in the schema.

Example 3.6.6. *Consider again the definitions for target relation hardWorking from Example 3.5.7 over the Original and 4NF UW-CSE schemas in Table 3.1. Let the INDs student[stud] = inPhase[stud] and student[stud] = yearsInProgram[stud] hold in the Original schema. Assume that Castor wants to generalize these clauses to cover example e', which satisfies student(x) but does not satisfy inPhase$(x,$'prelim'$)$. Castor removes the inPhase literal from the first clause and then removes literals with relation symbols student and yearsInProgram due to the INDs in the Original schema. It also removes student$(x,$'prelim','3'$)$ from the second clause. Over both schemas, the resulting*

generalization is:

$$hardWorking(x) \leftarrow publication(z, x).$$

Lemma 3.6.7. *Castor's ARMG algorithm is schema independent.*

Proof. Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a bijective transformation that performs compositions or decompositions from relations in schema $\mathcal{R} = (\mathbf{R}, \Sigma_{\mathcal{R}})$ to relations in schema $\mathcal{S} = (\mathbf{S}, \Sigma_{\mathcal{S}})$. Assume that τ decomposes relation $R_i \in \mathbf{R}$ to relations $S_{i_1} \dots S_{i_m} \in \mathbf{S}$. Assume that the input to the $ARMG$ algorithm over schema $\mathcal{R}$ is the bottom-clause for seed example e, denoted as $\overrightarrow{C_{\mathcal{R}}}$, which has the form of $T(w) \leftarrow L_1(u_1), \cdots, L_n(u_n)$. The input to the algorithm over schema $\mathcal{S}$ is the bottom-clause for seed example e, denoted as $\overrightarrow{C_{\mathcal{S}}}$, which has the form $T(w) \leftarrow P_1(v_1), \cdots, P_k(v_k)$. Clauses $\overrightarrow{C_{\mathcal{R}}}$ and $\overrightarrow{C_{\mathcal{S}}}$ are generated by the Castor bottom-clause construction algorithm and according to Lemma 3.6.5 are equivalent. They also do not contain any redundant literals.

The mapping between equivalent clauses over $\mathcal{R}$ and $\mathcal{S}$, δ_τ, that is associated with τ projects each literal with relation symbol R_i in $\overrightarrow{C_{\mathcal{R}}}$ to literals with relation symbols $S_{i_1} \dots S_{i_m}$ in $\overrightarrow{C_{\mathcal{S}}}$. Hence, there is a bijective mapping M that maps each literal $R_i(u_l)$ in the body of $\overrightarrow{C_{\mathcal{R}}}$ to a set of literals $S_{i_1}(v_j) \dots S_{i_m}(v_{j+(i_m-i_1)})$ in the body of $\overrightarrow{C_{\mathcal{S}}}$. According to Lemma 3.6.5, ordered clauses $\overrightarrow{C_{\mathcal{R}}}$ and $\overrightarrow{C_{\mathcal{S}}}$ are equivalent. Therefore, a literal L_l appears before L_o in the body of $\overrightarrow{C_{\mathcal{R}}}$ if and only if all literals in $M(L_l)$ appear before the ones in $M(L_o)$ in $\overrightarrow{C_{\mathcal{S}}}$. The mapping δ_τ only projects each literal with relation symbol $R_i(u_l)$ to a set of literals in $M(R_i(u_l))$. Hence, the free tuples in every pair of literals L_l and L_o in $\overrightarrow{C_{\mathcal{R}}}$ have a variable in common if and only if the sets of free tuples in $M(L_l)$ and $M(L_o)$ have a shared variable. Otherwise, $\overrightarrow{C_{\mathcal{R}}}$ and $\overrightarrow{C_{\mathcal{S}}}$ are not equivalent.

Assume that Castor removes literal L_b in $\overrightarrow{C_{\mathcal{R}}}$ because it is the blocking atom in the current iteration. Let the positive example considered for this iteration of the algorithm be e'. If L_b is the blocking atom, the sub-clause of $\overrightarrow{C_{\mathcal{R}}}$ up to and excluding L_b covers e' and the one up to and including L_b does not cover e'. Because mapping M preserves the order of literals, the sub-clause of $\overrightarrow{C_{\mathcal{S}}}$ up to and excluding literals in $M(L_b)$ covers e' and the one up to and including literals in $M(L_b)$ does not cover it. Hence, at least one literal in $M(L_b)$ is a blocking atom in $\overrightarrow{C_{\mathcal{S}}}$. If the algorithm removes this literal, it also drops the rest of literals in $M(L_b)$ because the free tuples of these literals do not satisfy the IND between relation symbols of $M(L_b)$ in the canonical database instance of $\overrightarrow{C_{\mathcal{S}}}$

after removing the blocking atom in $\overrightarrow{C_{\mathcal{S}}}$. Similarly, if one of the literals in $M(L_b)$ is a blocking atom, L_b will be also a blocking atom. In this case, the *ARMG* algorithm will also remove the non-blocking atoms in $M(L_b)$ that are not members of $M(L_o)$, $L_b \neq L_o$, as they do not satisfy any IND after removing the blocking atom.

Assume that a literal L_l is removed because it does not satisfy any IND in the canonical database instance of $\overrightarrow{C_{\mathcal{R}}}$ immediately after dropping the blocking atom L_b. Let Σ_1 be an IND between the relation symbol of L_b and the relation symbol of L_l. Because τ preserves the INDs between relations in $\mathbf{R}$, there is also an IND Σ_2 between the relation symbol of a literal P_l in $M(L_l)$ and the relation symbol of a literal in $M(L_b)$. Because L_b is a blocking atom, *ARMG* algorithm has already removed all literals in $M(L_b)$ from $\overrightarrow{C_{\mathcal{S}}}$. Assume that the free tuples of P_l and another literal P_o in $\overrightarrow{C_{\mathcal{S}}}$ satisfy Σ_2. If P_o has not been already removed from $\overrightarrow{C_{\mathcal{S}}}$, the free tuples of L_l and L_o satisfy the IND constraint Σ_1 in the canonical database of $\overrightarrow{C_{\mathcal{R}}}$. Thus, L_l should not have been removed from $\overrightarrow{C_{\mathcal{R}}}$. Therefore, P_o is removed from $\overrightarrow{C_{\mathcal{S}}}$. Hence, P_l must also be removed from $\overrightarrow{C_{\mathcal{S}}}$ as it does not satisfy any IND. After removing P_l, all literals in $M(L_l)$ will be removed from $\overrightarrow{C_{\mathcal{S}}}$. Using similar argument, the *ARMG* algorithm removes a literal L_r from $\overrightarrow{C_{\mathcal{R}}}$ because its free tuple does not satisfy any IND after dropping another literal, the algorithm removes the literals in $M(L_r)$ that are not members of $M(L_o)$, $L_r \neq L_o$. If the algorithm eliminates a literal P_r from $\overrightarrow{C_{\mathcal{S}}}$ because its free tuple does not satisfy any IND, the algorithm also removes the literals L_r, where $P_r \in M(L_r)$, from $\overrightarrow{C_{\mathcal{R}}}$. Finally, if the algorithm removes a literal because it is not head-connected, it also removes its corresponding literals over the decomposition and vice versa. □

3.6.2.2 Negative Reduction

Castor further generalizes clauses produced by *ARMG* by removing literals from clauses using a step called *negative reduction.* Removing literals from a clause results in a more general clause, which may cover more positive and negative examples than the original clause. In negative reduction, Castor only removes non-essential literals. A literal is *non-essential* if after it is removed from a clause, the number of negative examples covered by the clause does not increase [48, 50]. Castor uses INDs with equality to compute equivalent reductions of clauses over composition and decomposition. Given a clause $\overrightarrow{C}$ and inclusion class $\mathbf{N} = \{S_i \mid 1 \leq i \leq m\}$ over schema $\mathcal{S}$, an instance $Y_{\mathbf{N}}$ of $\mathbf{N}$

Algorithm 6: Castor's negative reduction algorithm.

Input : Clause $\overrightarrow{C} = T \leftarrow L_1, \cdots, L_n$, database instance I, negative examples E^-.
Output: Reduced clause $\overrightarrow{C'}$.
$E_c^- \leftarrow$ subset of E^- covered by $\overrightarrow{C}$
$\mathbf{I} \leftarrow$ list containing all instances of inclusion classes in $\overrightarrow{C}$
while *true* **do**
  $I_i \leftarrow$ first inclusion instance in $\mathbf{I}$ such that clause $T \leftarrow B$, where B contains literals in inclusion instances $I_1, \cdots, I_i$, has negative coverage E_c^-
  $\mathbf{H} \leftarrow$ inclusion instances in $\mathbf{I}$ that connect I_i with T
  $\mathbf{N} \leftarrow$ literals from inclusion instances $I_1, \cdots, I_i$ not in $\mathbf{H}$
  $\mathbf{I'} \leftarrow \mathbf{H} \cup [I_i] \cup \mathbf{N}$
  if $length(\mathbf{I'}) = length(\mathbf{I})$ **then**
    $C' = T \leftarrow B$, where B contains all literals in $\mathbf{I'}$
    Return C'
  $\mathbf{I} \leftarrow \mathbf{I'}$

is a set of literals $S_1(u_1), \cdots, S_m(u_m)$ in $\overrightarrow{C}$ such that for every IND $S_i[X] = S_j[X]$, $1 \leq i, j \leq m$, there are literals $S_i(u_i)$ and $S_j(u_j)$ in $Y_{\mathbf{N}}$ such that $\pi_X(u_i) = \pi_X(u_j)$. An instance $Y_{\mathbf{N}}$ over a clause $\overrightarrow{C}$ is *non-essential* if after removing all literals in $Y_{\mathbf{N}}$ from $\overrightarrow{C}$, the number of negative examples covered by the clause does not increase. First, for each literal L_j in the input clause $\overrightarrow{C}$, Castor computes the instances of inclusion classes in $\overrightarrow{C}$ that start with L_j. It creates a list containing all found instances, in the order in which they are found. Then, it iteratively removes non-essential instances from this list. In each iteration, it finds the first inclusion instance Y_i such that the sub-clause of $\overrightarrow{C}$ that contains all literals in every inclusion instance up to Y_i has the same negative coverage as $\overrightarrow{C}$. A *head-connecting inclusion instance* for Y_i contains literals that connect a literal in Y_i to the head of the clause by a chain of variables. Castor moves Y_i and its head-connecting inclusion instances to the beginning of the list, and discards the inclusion instances after Y_i. These instances can be discarded because they are non-essential. The algorithm iterates until the number of inclusion instances in the clause does not change after one iteration. At the end, it creates a clause whose head literal is the same as $\overrightarrow{C}$ and body contains all literals in the remaining instances of inclusion classes. Because negative reduction only removes literals from the clause, it does not decrease

the number of positive examples covered by the clause. Castor's negative reduction algorithm is depicted in Algorithm 6.

Lemma 3.6.8. *Castor's negative reduction algorithm is schema independent.*

Proof. Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a bijective transformation that performs compositions or decompositions from relations in schema $\mathcal{R} = (\mathbf{R}, \Sigma_{\mathcal{R}})$ to relations in schema $\mathcal{S} = (\mathbf{S}, \Sigma_{\mathcal{S}})$. Let $R \in \mathbf{R}$ and τ decompose relation R to relations $S_1, \cdots, S_m$, $1 \leq m \leq |\mathbf{S}|$. Let $\mathbf{N}$ be the inclusion class in $\Sigma_{\mathcal{S}}$ that contains relations $S_1, \cdots, S_m$. Assume that $\overrightarrow{C_{\mathcal{R}}}$ is a clause over schema $\mathcal{R}$ and contains k literals $R(u_i)$, $1 \leq i \leq k$. Let $\overrightarrow{C_{\mathcal{S}}}$ be the equivalent clause of $\overrightarrow{C_{\mathcal{R}}}$ over $\mathcal{S}$. Let *Reduce*(C) be the function that performs negative reduction on clause C.

Because $\overrightarrow{C_{\mathcal{R}}}$ contains k literals $R(u_i)$, $1 \leq i \leq k$, and $\overrightarrow{C_{\mathcal{R}}} \equiv \overrightarrow{C_{\mathcal{S}}}$, then $\overrightarrow{C_{\mathcal{S}}}$ must contain k instances of inclusion class $\mathbf{N}$. These instances of the inclusion class might or might not share literals. Let n be the number of instances of inclusion class $\mathbf{N}$ in $\overrightarrow{C_{\mathcal{S}}}$ that share literals. Without loss of generality, we assume that instances can only share the first literal. That is, instances $I_{\mathbf{N}i}$ and $I_{\mathbf{N}j}$ share a literal if they have the form $I_{\mathbf{N}i} = S_1(v_{\text{_}1}), S_2(v_{i2}), \cdots, S_m(v_{im})$ and $I_{\mathbf{N}j} = S_1(v_{\text{_}1}), S_2(v_{j2}), \cdots, S_m(v_{jm})$. We prove that *Reduce*$(\overrightarrow{C_{\mathcal{R}}}) \equiv$ *Reduce*$(\overrightarrow{C_{\mathcal{S}}})$ by induction on n.

Base case: let $n = 1$. Clause $\overrightarrow{C_{\mathcal{R}}}$ contains literal $R(u)$ and $\overrightarrow{C_{\mathcal{S}}}$ contains an instance of inclusion class $\mathbf{N}$ with literals $S_1(v_1), \cdots, S_m(v_m)$ such that $\bowtie_{l=1}^{m} [v_l] = u$. Notice that $\overrightarrow{C_{\mathcal{R}}}$ may contain other literals with relation R and $\overrightarrow{C_{\mathcal{S}}}$ may contain other instances of inclusion class $\mathbf{N}$. However, because $n = 1$, these instances do not share literals and can be treated independently. Then, Castor removes literal $R(u)$ in $\overrightarrow{C_{\mathcal{R}}}$ if and only if it removes literals $S_1(v_1), \cdots, S_m(v_m)$ in $\overrightarrow{C_{\mathcal{S}}}$.

Assumption step: let $n = k$. Clause $\overrightarrow{C_{\mathcal{R}}}$ contains literals $[R(u_i)]$, $1 \leq i \leq k$, clause $\overrightarrow{C_{\mathcal{S}}}$ contains literals $S_1(v_{\text{_}1})$, $[S_2(v_{i2}), \cdots, S_m(v_{im})]$, $1 \leq i \leq k$, and $\overrightarrow{C_{\mathcal{R}}} \equiv \overrightarrow{C_{\mathcal{S}}}$.

Induction step: let $n = k+1$. Let $\overrightarrow{C_{\mathcal{S}}}$ contain $k+1$ instances of inclusion class $\mathbf{N}$, which share the first literal. Let $\overrightarrow{C_{\mathcal{R}}}$ be the equivalent clause, which contains $k+1$ literals $R(u_i)$, $1 \leq i \leq k+1$. We divide instances in $\overrightarrow{C_{\mathcal{S}}}$ in two: $\mathbf{I_{N(1..k)}} = S_1(v_{\text{_}1}), [S_2(v_{i2}), \cdots, S_m(v_{im})]$, $1 \leq i \leq k$ and $I_{\mathbf{N}(k+1)} = S_1(v_{\text{_}1}), S_2(v_{(k+1)2}), \cdots, S_m(v_{(k+1)m})$. We also divide literals in $\overrightarrow{C_{\mathcal{R}}}$ in two: $\mathbf{R_{1..k}} = [R(u_i)]$, $1 \leq i \leq k$, and $R(u_{k+1})$. Let $\overrightarrow{C'_{\mathcal{S}}}$ contain all literals in $\mathbf{I_{N(1..k)}}$ and $\overrightarrow{C'_{\mathcal{R}}}$ contain all literals in $\mathbf{R_{1..k}}$. We examine the cases where we add literal $R(u_{k+1})$ to $\overrightarrow{C'_{\mathcal{R}}}$ such that $\overrightarrow{C'_{\mathcal{R}}} \cup \{R(u_{k+1})\} = \overrightarrow{C_{\mathcal{R}}}$, and we add all literals in

instance $I_{\mathbf{N}(k+1)}$ to $\overrightarrow{C'_{\mathcal{S}}}$ such that $\overrightarrow{C'_{\mathcal{S}}} \cup I_{\mathbf{N}(k+1)} = \overrightarrow{C_{\mathcal{S}}}$.

Case 1: Castor removes all literals in $\mathbf{R_{1..k}}$ and literal $R(u_{k+1})$ if and only if it removes all literals in $\mathbf{I_{N(1..k)}}$ and $I_{\mathbf{N}(k+1)}$. Then, $Reduce(\overrightarrow{C_{\mathcal{R}}}) \equiv Reduce(\overrightarrow{C_{\mathcal{S}}})$.

Case 2: Castor removes all literals in $\mathbf{R_{1..k}}$ but not literal $R(u_{k+1})$ if and only if it removes all literals in $\mathbf{I_{N(1..k)}}$, but not literals in $I_{\mathbf{N}(k+1)}$. Notice that literal $S_1(v_{_1})$ stays in clause $Reduce(\overrightarrow{C_{\mathcal{S}}})$ because it is in instance $I_{\mathbf{N}(k+1)}$. Because $\tau(R(u_{k+1})) = S_1(v_{_1})$, $S_2(v_{(k+1)2}), \cdots, S_m(v_{(k+1)m})$, then $Reduce(\overrightarrow{C_{\mathcal{R}}}) \equiv Reduce(\overrightarrow{C_{\mathcal{S}}})$.

Case 3: Castor removes literal $R(u_{k+1})$ but not literals in $\mathbf{R_{1..k}}$ if and only if it removes all literals in $I_{\mathbf{N}(k+1)}$, but not literals in $\mathbf{I_{N(1..k)}}$. Again, notice that literal $S_1(v_{_1})$ stays in clause $Reduce(\overrightarrow{C_{\mathcal{S}}})$ because it is in instances $\mathbf{I_{N(1..k)}}$. Because we know that $Reduce(\overrightarrow{C'_{\mathcal{R}}}) \equiv Reduce(\overrightarrow{C'_{\mathcal{S}}})$ (assumption step), then $Reduce(\overrightarrow{C_{\mathcal{R}}}) \equiv Reduce(\overrightarrow{C_{\mathcal{S}}})$. □

Based on Lemmas 3.6.5, 3.6.7, and 3.6.8, Castor is schema independent.

3.6.3 Generating Safe Clauses

Let the *head-variables* of a clause be the ones that appear in its head literal. A clause is *safe* if every head-variable appears in some literal in the body of the clause. A definition is safe if all its clauses are safe. The results of safe clauses and definitions are finite over a (finite) database. By default, current relational learning algorithms, including Castor, may learn *unsafe* Datalog definitions [2]. Because an unsafe definition produces infinitely many answers over a (finite) database, it is *not* desirable in many relevant applications, such as learning database queries from examples [3, 44]. Furthermore, a relational learning algorithm that learns only safe clauses can learn a definition from positive examples only. An *empty clause* is the clause that does not have any literals in its body. An empty clause covers all positive and negative examples. By forcing the algorithm to learn only safe clauses, it must learn non-empty clauses. Therefore, with only positive examples as input, the algorithm would learn a definition that contains safe clauses and has the highest score. In this section, we describe how Castor can be modified to generate only safe definitions. As we have explained, Castor first constructs the bottom-clause associated with some positive example e, and then generalizes this clause using $ARMG$ and negative reduction.

Bottom-clause Construction: Bottom-clause construction uses the positive example e as the initial head-literal for the bottom-clause. Castor picks every literal in body of the bottom-clause based on the constants and variables in the head-literal. Thus, the bottom-clause is guaranteed to be safe.

Safe ARMG Algorithm: Let the *ARMG* algorithm take as input clause $\overrightarrow{C}$ and positive example e, and produce as output clause $\overrightarrow{C'}$. Clause $\overrightarrow{C'}$ may not be safe. Castor checks whether $\overrightarrow{C'}$ is safe. If $\overrightarrow{C'}$ is safe, Castor considers it as a candidate; otherwise, Castor simply ignores it.

Safe Negative Reduction: In negative reduction, Castor first computes all instances of inclusion classes, and then iteratively removes non-essential instances. In order to output a safe clause, Castor first sorts all instances of inclusion classes by the number of head-variables appearing in the instance in descending order. Then, in each iteration, Castor finds the first inclusion instance Y_i such that the sub-clause of $\overrightarrow{C}$ that contains all literals in every inclusion instance up to Y_i has the same negative coverage as $\overrightarrow{C}$. Castor then finds the head-connecting inclusion instances for Y_i. Let these instances be called $\mathbf{H_{Y_i}}$. Next, from the instances of inclusion classes that will be discarded, Castor finds the first instances that contain head-variables that do not appear in Y_i or $\mathbf{H_{Y_i}}$. Let these instances be $\mathbf{S_{Y_i}}$ The goal is to find literals needed to make the resulting clause safe. These literals are guaranteed to exist because the clauses produced by *ARMG* are forced to be safe. Castor then moves Y_i, $\mathbf{H_{Y_i}}$, and $\mathbf{S_{Y_i}}$ to the beginning of the list, and discards the inclusion instances after Y_i, except the ones in $\mathbf{S_{Y_i}}$. The algorithm continues its normal operation until the number of inclusion instances in the clause does not change. Finally, it creates a clause whose body contains all literals in the remaining instances of inclusion classes.

3.6.4 General Composition and Decomposition

Castor is robust over schema variations caused by bijective decompositions and compositions as defined in Section 3.3. Bijective compositions need at least one IND with equality in the original schema. Bijective decompositions need at least one IND with equality in the transformed schema. We have observed several examples of these transformations in real-world databases, some of which we report in Section 3.8. However, in addition to INDs with equality, schemas often have INDs in the general form of subset

or equality. One can use these INDs to define a more general decomposition. More precisely, a *general decomposition* of schema $\mathcal{R}$ with single relation symbol R is schema $\mathcal{S}$ with relation symbols $S_1 \ldots S_n$ that satisfies all conditions in Definition 3.3.1 but at least one IND in $\mathcal{S}$ (in the second condition of Definition 3.3.1) is an IND in the form of subset or equality. A general decomposition of a schema with multiple relations is the union of general decompositions over each relation symbol in the schema.

A general decomposition is invertible but not bijective [2]. Consider the general decomposition from $\mathcal{R} : \{R_1(A, B, C)\}$ to $\mathcal{S} : \{S_1(A, B), S_2(A, C)\}$ with IND $S_2[A] \subseteq S_1[A]$, and the instance $I_{\mathcal{S}}$ of $\mathcal{S} : I_{S_1} = \{(a_1, b_1), (a_2, b_2)\}$, $I_{S_2} = \{(a_1, c_1)\}$. There is *not* any instance of $\mathcal{R}$ that represents the same information as $I_{\mathcal{S}}$. Hence, it is not clear how to define schema independence for $I_{\mathcal{S}}$. Also, the composition from $\mathcal{S}$ to $\mathcal{R}$ is not invertible as $I_{S_1} \bowtie I_{S_2}$ loses tuple (a_2, b_2), which cannot be recovered. As some original and transformed databases in this composition do not have the same information, it is not reasonable to expect equivalent learned definitions over these databases.

One may resolve these issues by considering databases with labeled nulls, e.g., by using weak universal relation assumption [2, 27]. For example, one can compose instance $I_{\mathcal{S}}$ in the last example to $I_{\mathcal{R}} : \{(a_1, b1, c_1),\ (a_2, b2, x)\}$ where x is a labeled null that reflects the existence of an unknown value. However, using nulls requires defining the semantics of learning over databases with labeled nulls and schema independence over transformations that introduce labeled nulls, which is not trivial. Instead, we define schema independence for general decompositions by ignoring the instances in the transformed schema that do not have any corresponding instance in the original schema. We ignore these instances because it is not reasonable to expect Castor or any other learning algorithm to learn equivalent definitions over instances where the transformation between the schemas is not bijective. Hence, we only consider the cases where the mapping between the instances in the original and the remaining instances of the transformed schemas is bijective. Therefore, the transformation between schemas where we only consider these instances is also definition-bijective. We define hypothesis-invariance and schema independence as defined in Section 3.2 for this mapping. An algorithm is schema independent over a general decomposition if it is schema independent over its mapping between the corresponding instances of the original and decomposed schemas.

A *general composition* is the inverse of a general decomposition. As we have shown, general compositions lose information. Thus, it is not reasonable to expect algorithms

to be schema independent over them. We limit the instances of the original schema so that the general composition becomes invertible. For simplicity, we define schema independence for a general composition whose transformed schema has a single relation. Our definition extends for schemas with multiple relations. Let schema $\mathcal{R}$ with a single relation symbol R be a general composition of schema $\mathcal{S}$ with relation symbols $S_1 \ldots S_n$ such that for all S_i, S_j, $1 \leq i, j \leq n$, $X = sort(S_i) \cap sort(S_j) \neq \emptyset$, $\mathcal{S}$ has IND $S_i[X] \subseteq S_j[X]$. The natural join between $S_1 \ldots S_n$ does not lose any tuples in an instance $I_\mathcal{S}$ of $\mathcal{S}$ if and only if for each IND $S_i[X] \subseteq S_j[X]$ in $\mathcal{S}$ we have $\pi_X(I_{S_i}) = \pi_X(I_{S_j})$, where I_{S_i} and I_{S_j} are relations of S_i and S_j in $I_\mathcal{S}$, respectively. Let $J(\mathcal{S})$ denote instances with the aforementioned property in $\mathcal{S}$. The mapping from $J(\mathcal{S})$ to $I(\mathcal{R})$ is bijective. Therefore, the transformation between $\mathcal{R}$ and $\mathcal{S}$ is definition-bijective. Thus, hypothesis-invariance and schema independence properties in Section 3.2 can be defined for this mapping. An algorithm over the general composition from $\mathcal{S}$ to $\mathcal{R}$ is *schema independent* if it is schema independent over the mapping between $J(\mathcal{S})$ to $I(\mathcal{R})$. We call a finite application of general decompositions and compositions a general decomposition and composition. An algorithm is schema independent over a general decomposition and composition if it is schema independent over its general decompositions and general compositions.

Consider again schema $\mathcal{S}$ with relation symbols $S_1 \ldots S_n$. To achieve schema independence over general composition and decomposition, given instance $I_\mathcal{S} \in J(\mathcal{S})$, Castor finds each IND $S_i[X] \subseteq S_j[X]$ in $\mathcal{S}$ where $\pi_X(I_{S_i}) = \pi_X(I_{S_j})$ and adds the IND to its list of IND with equality in a preprocessing step. It then proceeds to its normal execution. The proofs of Lemmas 3.6.5, 3.6.7, and 3.6.8 extend for the corresponding instances of $\mathcal{R}$ and $\mathcal{S}$ that have the same information in non-bijective decompositions. Using a similar argument, these proofs also hold for the corresponding instances that have the same information over general decomposition. Thus, Castor is schema independent over general decompositions and compositions. Using this method, Castor also handles combinations of INDs in general form and INDs with equality.

The pre-processing step of checking for each IND $S_i[X] \subseteq S_j[X]$ in schema $\mathcal{S}$ whether $\pi_X(I_{S_i}) = \pi_X(I_{S_j})$ holds may take a long time and some users may not want to wait for this pre-processing phase to finish. Another approach is to use INDs in the form of subset or equality in Castor directly as follows. We extend Castor to use both INDs with equality and in general form. In the rest of this section, we refer to both type of INDs simply as

INDs and denote them by the symbol $\subseteq$. We redefine an inclusion class $\mathbf{N}$ in schema $\mathcal{S}$ as a set of relation symbols S_i, S_j in $\mathcal{S}$ such that there is a sequence of INDs $S_k[X_k] \subseteq S'_k[X_k]$ or $S'_k[X_k] \subseteq S_k[X_k]$ $i \leq k \leq j$, in $\mathcal{S}$ where $X_k = sort(S_k) \cap sort(S'_k)$ and $S_{k+1} = S'_k$ for $i \leq k \leq j-1$. Assume that Castor picks a tuple s_i from relation S_i in inclusion class $\mathbf{N}$ during the bottom-clause construction. For each $S_i[X] \subseteq S_j[X]$ in $\mathbf{N}$, Castor selects all tuples s_j of relation I_{S_j}, $i \neq j$ such that $\pi_X(s_j) \subseteq \pi_X(s_i)$. Castor repeats this process for s_j until it exhausts all INDs in $\mathbf{N}$. After this step, Castor follows the bottom-clause construction algorithm explained in Section 3.6.1. Since the natural join between relations in $\mathcal{S}$ is acyclic, the pairwise consistency implies the global consistency of the join tuples. For the same reason, the proof of Proposition 3.6.4 extends for INDs. Hence, the INDs in each inclusion class are not cyclic and Castor efficiently finds the tuples that join according to the INDs. We also extend Castor's *ARMG* algorithm to ensure that the free tuple of each literal $S(u)$, u, satisfies all INDs in which S participates after a blocking atom is removed. If u does not satisfy any of its corresponding INDs, $S(u)$ it is removed from the clause. Finally, we redefine the instance of an inclusion class $\mathbf{N}$, $Y_{\mathbf{N}}$, in an ordered clause $\overrightarrow{C}$ as a set of literals $S_1(u_1), \cdots, S_m(u_m)$ in $\overrightarrow{C}$ such that for each IND $S_i[X] \subseteq S_j[X]$, $1 \leq i, j \leq m$, there are literals $S_i(u_i)$ and $S_j(u_j)$ in $Y_{\mathbf{N}}$ where $\pi_X(u_i) = \pi_X(u_j)$. We modify our negative reduction algorithm in Section 3.6.2.2 to use the new definition of inclusion class instance.

The extension of Castor that uses INDs in the form of subset may not be schema independent, as it may miss some tuples in bottom-clause construction or ignore some literals in the *ARMG* algorithm. For example, consider the general decomposition from $\mathcal{R}$:$\{R_1(A, B, C)\}$ to $\mathcal{S}$:$\{S_1(A, B), S_2(A, C)\}$ with IND $S_2[A] \subseteq S_1[A]$ and instances $J_{\mathcal{R}}$: $J_{R_1} = \{(a_1, b_1, c_1)\}$ and $J_{\mathcal{S}}$: $J_{S_1} = \{(a_1, b_1)\}$, $J_{S_2} = \{(a_1, c_1)\}$. Assume that the modified Castor bottom-clause construction over $J_{\mathcal{S}}$ starts with tuple (a_1, b_1). IND $S_2[A] \subseteq S_1[A]$ does not force Castor to select (a_1, c_1) for the bottom-clause. Hence, Castor delivers non-equivalent bottom-clauses over $J_{\mathcal{S}}$ and $J_{\mathcal{R}}$. Our empirical results in Section 3.8 show that this extension of Castor is more robust than other algorithms over general decomposition and composition.

3.6.5 Castor System Design Choices and Implementation

Current bottom-up algorithms do not run efficiently over medium or large databases because they produce long bottom-clauses [50]. Also, these clauses are time-consuming to evaluate. A relational learning algorithm evaluates a clause by computing the number of positive and negative examples covered by the clause. These tests dominate the time for learning [22]. It is generally time-consuming to evaluate clauses with many literals. Castor implements several optimizations to run efficiently over large databases.

In-memory RDBMS: Castor is implemented on top of the in-memory relational database management system (RDBMS) VoltDB (*voltdb.com*). Relational databases are usually stored in RDBMSs. Therefore, it is natural to implement a relational learning algorithm on top of an RDBMS. Castor performs bottom-clause construction multiple times during the learning process. The bottom-clause construction algorithm queries the database multiple times, each of which selects all tuples in a table that match given constants from the training data. We leverage RDBMS indexing to improve the running time of these queries.

Stored Procedures: We implement the bottom-clause construction algorithm inside a stored procedure to reduce the number of API calls made from Castor to the RDBMS. Castor makes only one API call per each bottom-clause. The first time that Castor is run on a schema, it creates the stored procedure that implements the bottom-clause construction algorithm for the given schema. Castor reuses the stored procedure when the algorithm is run again, with either new training data or an updated database instance.

Efficient Clause Evaluation: One approach to computing the number of positive (negative) examples covered by a clause is to join the table containing the positive (negative) examples with the tables corresponding to all literals in the body of the clause. If two literals share a variable, then a natural join between the two columns corresponding to the shared variable in the literals is used. This strategy works well when clauses are short, as in top-down algorithms [67]. However, our empirical studies show that the time and space requirements for this approach are prohibitively large on large clauses generated by bottom-up algorithms. Thus, we perform coverage tests by using a subsumption engine. A *ground bottom-clause* is a bottom-clause that only contains constants. A candidate clause C covers example e if and only if C θ-subsumes

the ground bottom-clause $\perp_e$ associated with e. Castor uses the subsumption engine Resumer2 [43]. Resumer2 efficiently checks if clause C covers example e by deciding the subsumption between C and the ground bottom-clause $\perp_e$ of e. Given clause C and a set of examples E, Castor checks if C covers each $e \in E$ separately. Castor divides E in subsets and performs coverage testing for each subset in parallel.

Coverage Tests: Castor optimizes the generalization process by reducing the number of coverage tests. Castor first generates the bottom-clause relative to a positive example. Then, Castor generalizes this clause. If clause C covers example e, then clause C'', which is more general than C, also covers e. If Castor knows that C covers e, it does not check if C'' covers e.

Minimizing Clauses: Bottom-up algorithms such as Castor produce large clauses, which are expensive to evaluate. Castor minimizes bottom-clauses by removing syntactically redundant literals. A literal L in clause C is *redundant* if C is equivalent to $C' = C - \{L\}$. Clause equivalence between C and C' can be determined by checking whether C θ-subsumes C' and C' θ-subsumes C. By definition, C' θ-subsumes C because C' is a subset of C. Therefore, we only need to check whether C θ-subsumes C'. Castor minimizes clauses using theta-transformation [19]. It uses a polynomial-time approximation of the clausal-subsumption test, which is efficient and retains the property of correctness. Given clause C, for each literal L in C, the algorithm checks if $C \subseteq C' = C - \{L\}$. If this holds, then L is redundant and will be removed. Minimizing bottom-clauses reduces the hypothesis space considered by Castor. It also makes coverage testing faster. Castor also minimizes learned clauses before adding them to the definition. Minimized clauses are more concise and interpretable.

3.7 Query-based algorithms

In this section, we consider query-based learning algorithms, which learn exact definitions by asking queries to an oracle [3, 7, 41, 58]. This type of algorithm has been used recently in various areas of database management, such as finding schema mappings and designing usable query interfaces [3, 14]. Queries can be of multiple types, however the most common types are equivalence queries and membership queries. In equivalence queries (EQ), the learner presents a definition to the oracle and the oracle returns *yes* if the definition is equal to the target relation definition; otherwise it returns a counter-

example. In membership queries (MQ), the learner asks if an example is a positive example, and the oracle answers *yes* or *no*.

Because query-based algorithms follow a different learning model, Definition 3.2.10 is not suited for evaluating their schema (in)dependence. Since a query-based algorithm can ask the oracle whether candidate definitions are correct, the algorithm will always learn the correct definitions by asking sufficient number of queries from the oracle. As it takes time and resources to answer queries, a desirable query-based algorithm should not ask too many queries [7]. For instance, some database query interfaces use query-based algorithms to discover users' intents [3]. Because the oracle for these algorithms is the user of the database, a more desirable algorithm should figure out the user's intent by asking fewer queries from the user.

Query-based algorithms are theoretically evaluated by their *query complexity* – the asymptotic number of queries asked by the algorithm [41]. Therefore, we analyze the impact of schema transformations on the query complexity of these algorithms. Generally, if an algorithm has different asymptotic behaviors over equivalent schemas, then the algorithm is schema dependent. One way to show that an algorithm has different asymptotic behaviors over different schemas is by comparing the lower bound on the query complexity of the algorithm against the upper bound on its query complexity. If the lower bound under one of the schemas is greater than the upper bound under another schema, then the algorithm is highly schema dependent. Of course, this behavior is not a desirable property, as it means that the choice of representation has a huge effect on the performance of the algorithm. However, we prove that a popular query-based algorithm called $A2$ suffers from this property.

$A2$ [41] is a query-based learning algorithm that learns function-free, first-order Horn expressions. The reasons for choosing this algorithm are three fold: i) $A2$ is representative of query-based learning algorithms that work on the relational model, ii) there is an implementation of the algorithm [7], iii) $A2$ is a generalization to the relational model of a classic query-based propositional algorithm [4].

Let $p_{\mathcal{R}}$ be the number of relations in schema $\mathcal{R}$ and $a_{\mathcal{R}}$ be the largest arity of any relation in schema $\mathcal{R}$. Let k be the largest number of variables in a clause, m be the number of clauses in the definition of the target relation, and n be the largest number of constants (i.e. objects) in any example. Parameters k, m, and n are independent of the schema. The upper bound on the number of EQs and MQs made by the $A2$

algorithm over schema $\mathcal{R}$ is $O(m^2(p_\mathcal{R})k^{(a_\mathcal{R})+3k} + nm(p_\mathcal{R})k^{(a_\mathcal{R})+k})$ and the lower bound is $\Omega(m(p_\mathcal{R})k^{(a_\mathcal{R})})$ [41].

Theorem 3.7.1. *There is a schema $\mathcal{R}$ and decomposition τ, where $\tau(\mathcal{R}) = \mathcal{S}$, such that $\Omega(m(p_\mathcal{R})k^{(a_\mathcal{R})}) > O(m^2(p_\mathcal{S})k^{(a_\mathcal{S})+3k} + nm(p_\mathcal{S})k^{(a_\mathcal{S})+k})$.*

Proof. Let schema $\mathcal{R} = (\mathbf{R}, \Sigma_\mathcal{R})$ contain the single relation $R(A_1, \cdots, A_l)$. Assume that $l \geq 2$ and there are $l-1$ functional dependencies $A_1 \rightarrow A_i$, $2 \leq i \leq l$, in $\Sigma_\mathcal{R}$. Let $\tau(\mathcal{R}) = \mathcal{S} = (\mathbf{S}, \Sigma_\mathcal{S})$ be a decomposition of $\mathcal{R}$, such that relation $R(A_1, \cdots, A_l) \in \mathbf{R}$ is decomposed into $l-1$ relations in $\mathbf{S}$ in the form of $S_i(A_1, A_i)$, $2 \leq i \leq l$. For each relation $S_i(A_1, A_i) \in \mathbf{S}$, $\Sigma_\mathcal{S}$ contains the functional dependency $A_1 \rightarrow A_i$. For each set of relations $S_i(A_1, A_i)$, $2 \leq i \leq l$, $\Sigma_\mathcal{S}$ also contains $2(l-1)$ inclusion dependencies in the form of $S_2[A_1] \subseteq S_j[A_1]$ and $S_j[A_1] \subseteq S_2[A_1]$, $2 < j \leq l$. Because the number of relations in $\mathcal{R}$ is $p_\mathcal{R} = 1$ and the maximum arity is $a_\mathcal{R}$, then the maximum number of relations in $\mathcal{S}$ is $p_\mathcal{S} = a_\mathcal{R} - 1$. We also have that $a_\mathcal{S} = 2$.

Let $\mathcal{L}_\mathcal{R}$ be the hypothesis language that consists of the subset of Horn definitions defined over schema $\mathcal{R}$ that contain a single clause in which no self-joins are allowed. All definitions in $\mathcal{L}_\mathcal{R}$ have the form $T(\mathbf{u}) \leftarrow R(x_1, x_2, \cdots, x_l)$, where T is the target relation and $\mathbf{u}$ is a subset of $\{x_1, x_2, \cdots, x_l\}$. Let δ_τ be the definition mapping for τ. Because transformation τ is a decomposition, for any definition $h_\mathcal{R} \in \mathcal{L}_\mathcal{R}$, its corresponding definition $\delta_\tau(h_\mathcal{R}) \in \mathcal{L}_\mathcal{S}$ also contains a single clause.

Any clause in a definition $h_\mathcal{R} \in \mathcal{L}_\mathcal{R}$ has at most l distinct variables, which corresponds to the arity of relation R. Therefore the largest number of variables in a clause is $k = l$. As schema $\mathcal{S}$ is a decomposition of schema $\mathcal{R}$, and no self-joins are allowed in $\mathcal{L}_\mathcal{R}$, the definition $h_\mathcal{S} = \delta(h_\mathcal{R}) \in \mathcal{L}_\mathcal{S}$ also has at most $k = l$ variables. Because definitions in $\mathcal{L}_\mathcal{R}$ and $\mathcal{L}_\mathcal{S}$ contain only one clause, then the maximum number of clauses in a definition is $m = 1$.

In order to prove our theorem, the following should hold for $\mathcal{R}$ and $\mathcal{S}$

$$\Omega(m(p_\mathcal{R})k^{(a_\mathcal{R})}) > O(m^2(p_\mathcal{S})k^{(a_\mathcal{S})+3k} + nm(p_\mathcal{S})k^{(a_\mathcal{S})+k})$$

where the left side of the inequality is the lower bound on the query complexity under schema $\mathcal{R}$ and the right side is the upper bound on the query complexity under schema $\mathcal{S}$. The operator $>$ means that $A2$ will always ask asymptotically more queries under

schema $\mathcal{R}$ than under schema $\mathcal{S}$. We have that k and m are the same for both schemas. We can also ignore n as it is independent of the hypothesis space and the schemas. Therefore, by canceling out some terms, the previous inequality can be rewritten as

$$\Omega(k^{(a_{\mathcal{R}})}) > O(m(a_{\mathcal{R}} - 1)k^{2+3k} + (a_{\mathcal{R}} - 1)k^{2+k}).$$

The first term in the upper bound dominates the second term, then we have

$$\Omega(k^{(a_{\mathcal{R}})}) > O(m(a_{\mathcal{R}} - 1)k^{2+3k})$$

Assuming that $m = 1$, as in $\mathcal{L}_{\mathcal{R}}$, we get

$$\Omega(k^{(a_{\mathcal{R}})}) > O((a_{\mathcal{R}} - 1)k^{2+3k})$$

This inequality holds for sufficiently large k and $a_{\mathcal{R}}$. □

The lower bound of $A2$ is the Vapnik-Chevonenkis dimension (VC-Dim) of the hypothesis language that consists of function-free, first-order Horn expressions. Therefore, we have proven in Theorem 3.7.1 that there are cases where the lower bound on the query complexity of *any* algorithm under this hypothesis language is greater than the upper bound on the query complexity of $A2$.

3.8 Experiments

We empirically evaluate several sample-based and query-based relational learning algorithms to answer the following questions:

1. Are existing relational learning algorithms schema independent? (Section 3.8.2)
2. Is Castor schema independent? (Section 3.8.2)
3. How does Castor's effectiveness and efficiency compare with other relational learning algorithms? (Section 3.8.2)
4. What is the relationship between the style of schema design and the effectiveness of a learning algorithm? (Section 3.8.2)

5. What is the effect of Castor's design choices? (Section 3.8.3)
6. What is the effect of the schema on the query complexity of query-based relational learning algorithms? (Section 3.8.4)

3.8.1 Experimental Settings

We use three datasets whose statistics are shown in Table 3.3. The **HIV-Large** dataset contains information about 42,000 chemical compounds obtained from the National Cancer Institute's AIDS antiviral screen (*wiki.nci.nih.gov/display/NCIDTPdata*). The schema contains relation *compound(comp, atm)*, which indicates that compound *comp* contains atom *atm*. It also has relations that indicate the chemical element that an atom represents, e.g., *element_C(atm)*, as well as relations to indicate properties of each atom, e.g., *p2_1(atm)*. The schema represents a bond between two atoms by relation *bonds(bd, atm1, atm2)*, and it has a relation for each type of a bond, e.g., *bondType1(bd, t1)*. The goal is to learn the relation *hivActive(compound)*, which indicates that *compound* has anti-HIV activity. The original HIV dataset is stored in flat files and does not have any information about its constraints. We explored the database for possible dependencies. In particular, we have discovered that the INDs *bonds[bd] = bondType1[bd], bonds[bd] = bondType2[bd], bonds[bd] = bondType3[bd]* hold in the database. We have used these dependencies to compose relations *bonds*, *bondType1*, *bondType2*, and *bondType3* into a single relation *bonds* and create a schema in 4NF, named 4NF-1. We also decompose relation *bonds* in the initial schema to relations *bondSource* and *bondTarget* to create another schema, called 4NF-2. The schemas and all INDs for this dataset are shown in Tables 3.4 and 3.5, respectively. In the **HIV-2K4K** dataset, we keep the same background knowledge, but reduce the number of examples to 2K positive and 4K negative examples.

The **UW-CSE** dataset contains information about an academic department and has been used as a benchmark in the relational learning literature [60]. The goal is to learn the target relation *advisedBy(stud, prof)*, as explained in Section 3.1 The dataset comes with a set of constraints in form of first-order logic clauses that should hold over the dataset domain. The INDs in these constraints are shown in Table 3.6 (top). If there are more INDs with equality in the schema, one can generate more schemas

Name	Schema	#R	#T	#P	#N
HIV-Large	Initial	80	14M	5.8K	36.8K
	4NF-1	77	7.8M		
	4NF-2	81	16M		
UW-CSE	Original	9	1.8K	102	204
	4NF	6	1.4K		
	Denormalized-1	5	1.3K		
	Denormalized-2	4	1.3K		
IMDb	JMDB	46	8.4M	1.85K	3.6K
	Stanford	41	10.5M		
	Denormalized	33	7.2M		

Table 3.3: Numbers of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset.

Initial	4NF-1	4NF-2
bonds(bd, atm1, atm2)	bonds(bd, atm1, atm2, t1, t2, t3)	bondSource(bd, atm1)
bondType1(bd, t1)		bondTarget(bd, atm2)
bondType2(bd, t2)		bondType1(bd, t1)
bondType3(bd, t3)		bondType2(bd, t2)
		bondType3(bd, t3)
	Common relations	
compound(comp, atm)	element_C(atm) ...	element_O(atm)
p2_0(atm)	p2_1(atm) ...	p3(atm)

Table 3.4: Schemas for the HIV-Large and HIV-2K4K datasets.

bonds[bd]=bondType1[bd]	bonds[bd]=bondType2[bd]
bonds[bd]=bondType3[bd]	
bonds[atm1]⊆compound[atm]	bonds[atm2]⊆ compound[atm]
elem_C[atm]⊆compound[atm] ...	elem_O[atm]⊆ compound[atm]
p2_0[atm]⊆compound[atm] ...	p3[atm]⊆compound[atm]

Table 3.5: INDs in the initial HIV dataset.

student[stud] = inPhase[stud]	yearsInProg[stud] ⊆ student[stud]
hasPosition[prof] = professor[prof]	ta[stud] ⊆ student[stud]
ta[crs] = taughtBy[crs]	
taughtBy[prof] = professor[prof]	student[stud] ⊆ yearsInProg[stud]
courseLevel[crs] = taughtBy[crs]	
inPhase[stud] ⊆ student[stud]	yearsInProg[stud] ⊆ student[stud]
hasPosition[prof] ⊆ professor[prof]	ta[stud] ⊆ student[stud]
taughtby[prof] ⊆ professor[prof]	taughtby[crs] ⊆ courseLevel[crs]

Table 3.6: INDs in the UW-CSE dataset. Top: INDs in the original dataset. Middle: INDs added to have bijective transformations. Bottom: INDs that should hold according to the semantics of the database.

from the original UW-CSE schema using composition transformation. To evaluate the effectiveness of algorithms over more varieties of schemas, we add the INDs shown in Table 3.6 (middle) to the schema. We enforce the constraints by removing a small fraction of tuples, 159 tuples, from the original dataset. We transform the original schema to three other different schemas. The original and a composed schema, called 4NF, are shown in Table 3.1. We compose *courseLevel* and *taughtBy* relations in 4NF schema to create a more denormalized schema, named Denormalized-1, and compose *courseLevel*, *taughtBy*, and *professor* in 4NF schema to generate the fourth schema, named Denormalized-2.

The **IMDb** (*imdb.com*) dataset contains information about movies. We learn the target relation *dramaDirector(director)*, which indicates that *director* has directed a drama movie. JMDB (*jmdb.de*) provides a relational database of IMDb data under a 4NF schema. We create a subset of JMDB database by selecting the movies produced after year 2000 and their related entities, e.g., actors, directors, producers. The relationships between relation *movie(id, title, year)* and its related relations, e.g., *director(id, name)*, are stored in relations *movies2X* where X is the name of the related entity set, e.g., *movies2director(id, directorid)*. The resulting database has 11 INDs with equality in the form of *movies2X[Xid] =X[id]*, e.g., *movies2director[directorid] = director[id]*. To test over more transformations, we have changed 5 INDs in the form of subset to INDs in the form of equality, e.g., *movies2X[id] ⊆ movie[id]* to *movies2X[id] = movie[id]*, by removing some tuples from the database. We use the first set of 11 INDs with equality to compose 11 pairs of relations in JMDB schema to create a new schema, called Denormalized. We use the second set of INDs with equality to compose 5 relations in JMDB

JMDB	Stanford
movie(id, title, year)	movie(id, title, year, genreid,
movies2genre(id, genreid)	colorid, prodcompid,
movies2color(id, colorid)	directorid, producerid)
movies2director(id, directorid)	
movies2producer(id, producerid)	
movies2prodcomp(id, prodcompid)	
Common relations	
language(id, language)	plot(id, plot)
country(id, country)	color(id, color)
business(id, text)	altversion(id, version)
runningtime(id, times)	prodcompany(id, name)
actor(id, name, sex)	editor(id, name)
director(id, name)	producer(id, name)
writer(id, name)	akaname(name, akaname)
akatitle(id, langid, title)	cinematgr(id, name)
biography(id, name, bio)	movies2misc(id, miscid)
composer(id, name)	costdesigner(id, name)
distributor(id, name)	rating(id, rank, votes)
genre(id, genre)	misc(id, name)
mpaarating(id, text)	technical(id, text)
proddesinger(id, name)	releasedate(id, countryid, date)
movies2actor(id, actorid, character)	movies2editor(id, editorid)
movies2writer(id, writerid)	movies2cinematgr(id, cinamtid)
movies2composer(id, composerid)	movies2costdes(id, costdesid)
movies2language(id, langid)	certificate(id, countryid, cert)
movies2proddes(id, proddesid)	movies2country(id, countryid)

Table 3.7: JMDB and Stanford schemas for the IMDb dataset. Relations in bottom are contained in both schemas.

Denormalized	
movie(id, title, year)	language(id, language)
movies2actor(id, actorid, name, character, sex)	plot(id, plot)
movies2color(id, colorid, color)	altversion(id, version)
movies2X(id, Xid, name) where	runningtime(id, times)
X= {writer, editor, composer,	prodcompany(id, name)
cinematgr, costdes, proddes,	country(id, country)
director, producer, misc}	akaname(name, akaname)
akatitle(id, langid, title)	biography(id, name, bio)
distributor(id, name)	rating(id, rank, votes)
genre(id, genre)	releasedate(id, countryid, date)
movies2language(id, langid)	certificate(id, countryid, cert)
mpaarating(id, text)	technical(id, text)
movies2country(id, countryid)	business(id, text)

Table 3.8: Denormalized schema for the IMDb dataset.

movies2X[id] = movie[id]
where X= {genre, color, prodcompany, producer, director}
movies2Y[Yid] = Y[id]
where Y= {actor, cinematagr, color, composer, costdes, director,
editor, misc, proddes, producer, writer}
Z[id] $\subseteq$ movie[id]
where Z={business, runningtime, altversion, certificate,
plot, rating, akatitle, distributor, releasedate,
technical, movies2actor, movies2country, movies2composer,
movies2writer, movies2costdes, movies2misc, movies2editor,
movies2cinematgr, movies2language, movies2proddes}
certificate[countryid] $\subseteq$ country[countryid]
releasedate[countryid] $\subseteq$ country[countryid]
akatitle[langid] $\subseteq$ language[langid]
movies2country[countryid] $\subseteq$ country[countryid]
movies2language[langid] $\subseteq$ language[langid]
movies2genre[genreid] $\subseteq$ genre[genreid]
movies2prodcompany[prdcompid] $\subseteq$ prodcompany[prdcompid]

Table 3.9: INDs in IMDb dataset.

schema, and create a schema called Stanford that follows a structure similar to the one used in the Stanford Movie DB (*infolab.stanford.edu/pub/movies*). The three schemas and the full list of INDs in IMDb data are shown in Tables 3.7, 3.8 and 3.9. In the UW-CSE and IMDb datasets, we generate negative examples by using the closed-world assumption, and then sample to obtain twice as many negative examples as positive examples.

We compare Castor to three relational learning systems: FOIL [57], Aleph [61], and GILPS [50]. The **FOIL** system implements the FOIL algorithm but does not scale to medium and large datasets. Therefore, we also emulate FOIL using Aleph by forcing Aleph to follow a greedy strategy and call it **Aleph-FOIL**. Aleph is a well known Inductive Logic Programming (ILP) system that implements Progol [47]. To differentiate the two variations of Aleph used in our experiment, we call the default implementation of Aleph **Aleph-Progol**. GILPS implements **ProGolem**, which is a bottom-up algorithm.

Aleph contains the parameter *clauselength*, which restricts the size of the learned clauses. Over HIV-Large and HIV-2K4K, the definition for the target relation must contain long clauses. With the default value of $clauselength = 4$, Aleph-FOIL and Aleph-Progol do not learn any clauses. Therefore, we set this parameter to have values of 10 and 15.

Machine learning algorithms usually require parameter tuning to run them successfully. We use the default parameter configuration for all systems, with some exceptions. Because we use noisy datasets, we must allow the algorithms to learn clauses that cover some negative examples. To limit the number of negative examples covered by any learned clause, we require that the ratio of positive to negative examples covered by a clause (precision) is at least 2 to 1. That is, the number of positive examples examples covered by a clause must be two times greater than or equal to the number of negative examples covered by the clause. In FOIL, this value is set with the *accur* parameter; in Aleph it is set with the *minacc* parameter; in ProGolem and Castor it is set with the *minprec* parameter. In FOIL, the only settings that we modify is *aaccur=0.67*. In Aleph, the settings that we modify are *minacc=0.67*, *minpos=2*, *noise=inf*, and *openlist=1* (only for Aleph-FOIL). In Castor and ProGolem, the settings are *minprec=0.67*, *noise=1*, *minpos=2*, and *sample=1*, *beamwidth=1* for HIV-Large, HIV-2K4K, and IMDb, and *sample=20*, *beamwidth=3* for UW-CSE. In the IMDb dataset, we also restrict the number of literals with the same relation symbol added to a ground bottom clause in

one iteration of the bottom clause construction algorithm. We set this value to 10. If this value is unrestricted, a bottom clause may contain hundreds or thousands of literals with the same relation symbol (one for each tuple).

There are far fewer query-based relational learning systems available than the ones that use samples for learning. To empirically evaluate the schema independence of query-based learning methods, we use the **LogAn-H** system [7], which is an implementation of the A2 algorithm [41]. We call the learning algorithms that use batches of training samples, e.g., FOIL and ProGolem, sample-based algorithms to distinguish them from query-based algorithms in this section.

We refer to the quality of a definition as the *effectiveness* of the definition. We use the metrics of *precision* and *recall* to measure the effectiveness of definitions. Let the set of *true positives* for a definition be the set of positive examples in the testing data that are covered by the definition. The precision of a definition is the proportion of its true positives over all examples covered by the definition. The recall of a definition is the number of its true positives divided by the total number of positive examples in the testing data. Precision and recall are between 0 and 1, where an ideal definition delivers both precision and recall of 1. Similar to other machine learning tasks, it is not often possible to learn an ideal definition for a target concept due to various reasons, such as the hardness of the target concept, the lack of sufficient amount of training data, or simply because the target concept might not be definable. In these situations, the values of reasonable precision and recall for a definition depend on the underlying applications, e.g., 5% improvement in precision may not be important in a financial application but vital in a medical application. Nevertheless, definitions with higher precision or recall are generally more desirable [50, 57, 61]. We use the learning time to measure the *efficiency* of all systems. We perform 5-fold cross validation for UW-CSE and 10-fold cross validation for HIV and IMDb datasets. We evaluate precision, recall, and learning time, showing the average over the cross validation.

We ran experiments on a server containing 32 2.6GHz Intel Xeon E5-2640 processors, running CentOS Linux 7.2 with 50GB of main memory.

3.8.2 Sample-based Algorithms

Castor is schema independent over all datasets and delivers equal precision and recall across all schemas of each dataset in our experiments. However, other algorithms are schema dependent.

HIV datasets. Aleph-FOIL, Aleph-Progol and Castor are the only algorithms that scale to the HIV-2K4K dataset. Aleph-FOIL and Castor also scale to the HIV-Large dataset. The definitions learned by Aleph-FOIL and Aleph-Progol over different schemas are not equivalent as shown by their precision and recall values across schemas in Table 3.10. Different schemas cause Aleph-FOIL and Aleph-Progol to explore different regions of the hypothesis space. Aleph-FOIL and Aleph-Progol are not able to find any definition over the 4NF-2 schema of HIV-Large and HIV-2K4K datasets. The reason is that any good clause must contain information about bonds. In the 4NF-2 schema, this information is represented by two relations, *bondSource* and *bondTarget*, and three more to indicate their types. With a top-down search, these algorithms are not able to find a clause that contains these relations. Aleph-FOIL terminates without learning anything and Aleph-Progol does not terminate after 75 hours. Aleph-Progol does not terminate after 75 hours over the 4NF-2 schema of HIV-2K4K. FOIL crashes on both HIV datasets. ProGolem does not learn anything after 5 days running, even on smaller subsets of the HIV dataset.

UW-CSE dataset. As shown in Table 3.11, all algorithms except for Castor are schema dependent and learn non-equivalent definitions over different schemas of UW-CSE. As this dataset is smaller than the HIV and IMDb datasets, it has a relatively smaller hypothesis space. Hence, the degree of schema dependence for these algorithms over this dataset is generally lower than other datasets. The precision and recall of all algorithms are not significantly different across schemas. Over denormalized schemas, Aleph-FOIL learns definitions consisting of many clauses, each covering a few examples. This results in low generalization, hence very low precision and recall. On the other hand, over the Original schema, it learns definitions consisting of a lower number of clauses, each covering a greater number of examples. Note that Aleph-FOIL does not exactly emulate FOIL. FOIL uses a different evaluation function and explores an unre-

stricted hypothesis space. Therefore, FOIL does not suffer from the same problems as Aleph-FOIL. However, it is less effective than other algorithms. Castor's effectiveness is comparable to Aleph-Progol and ProGolem over the Original and 4NF schemas. Nevertheless, Aleph-Progol and ProGolem perform worse on other schemas. On the other hand, Castor is effective over all schemas.

IMDb dataset. The target relation for the IMDb dataset has an exact Datalog definition given the background knowledge and training examples. Castor finds this definition over all schemas and obtains precision and recall of 1, as shown in Table 3.12. Aleph-FOIL fails to find this definition over all schemas. Aleph-Progol finds this definition only over the Stanford schema. The definitions learned by Aleph-FOIL and Aleph-Progol over different schemas are largely different.

Relationship between style of design and effectiveness. Our results show that there is not any single style of design, e.g., 4NF, on which all algorithms, except for Castor, are effective over all datasets. Generally, the style of design on which a relational learning algorithm delivers its most effective results varies based on the metric of effectiveness, the dataset, and the algorithm. For example, Aleph-Progol delivers its highest precision over a denormalized schema, Denormalized-1, for UW-CSE, but its highest recall over the original schema, which is more normalized than 4NF. Aleph-Progol also delivers its lowest precision on UW-CSE data over another denormalized schema, Denormalized-2, for this dataset. Hence, it is generally hard to find a straightforward relationship between the style of design and the precision or recall of an algorithm over a given dataset. Furthermore, each algorithm prefers a different style of design over each dataset. For example, Aleph-Progol has higher overall precision and recall on the most normalized schema, original schema, for UW-CSE. But, it delivers its highest overall precision and recall over the most denormalized schema, Stanford, for IMDb. Finally, different algorithms prefer distinct styles of design over the same dataset. For example, FOIL delivers both its highest precision and highest recall over a denormalized schema for UW-CSE data, Denormalized-2, over which Aleph-Progol delivers both its lowest precision and lowest recall. Over the same database, ProGolem achieves both its highest precision and highest recall for the most normalized schema, i.e., original schema.

HIV-Large				
Algorithm	Metric	Initial	4NF-1	4NF-2
Aleph-FOIL	Precision	0.58	0.72	0
($clauselength = 10$)	Recall	0.42	0.91	0
	Time (h)	3	0.9	6
Aleph-FOIL	Precision	0.68	0.68	0
($clauselength = 15$)	Recall	0.41	0.85	0
	Time (h)	11.7	3.7	47
	Precision	0.81	0.81	0.81
Castor	Recall	0.85	0.85	0.85
	Time (h)	3.5	1.9	56
HIV-2K4K				
Aleph-FOIL	Precision	0.72	0.78	0
($clauselength = 10$)	Recall	0.69	0.81	0
	Time (m)	6.2	7.9	20.6
Aleph-FOIL	Precision	0.70	0.78	0
($clauselength = 15$)	Recall	0.79	0.89	0
	Time (m)	6.72	7.07	122.2
Aleph-Progol	Precision	0.70	0.79	-
($clauselength = 10$)	Recall	0.85	0.90	-
	Time (m)	58.5	72.2	> 75 h
Aleph-Progol	Precision	0.72	0.75	-
($clauselength = 15$)	Recall	0.89	0.87	-
	Time (m)	155.51	13.56	> 75 h
	Precision	0.80	0.80	0.80
Castor	Recall	0.87	0.87	0.87
	Time (m)	15.1	6.5	335.5

Table 3.10: Results of learning relations over the HIV-Large and HIV-2K4K dataset.

Algorithm	Metric	Original	4NF	Den-1	Den-2
FOIL	Precision	0.84	0.79	0.77	0.85
	Recall	0.35	0.36	0.42	0.47
	Time (s)	18.7	20.84	30.72	30.64
Aleph-FOIL	Precision	0.78	0.50	0.36	0.19
	Recall	0.17	0.18	0.13	0.11
	Time (s)	3.5	4.3	14.8	398.1
Aleph-Progol	Precision	0.95	0.97	0.98	0.55
	Recall	0.54	0.45	0.36	0.29
	Time (s)	9.7	13.2	27.9	334.8
ProGolem	Precision	0.95	0.95	0.80	0.82
	Recall	0.54	0.54	0.48	0.48
	Time (s)	24.4	28.8	26.7	54.1
Castor	Precision	0.93	0.93	0.93	0.93
	Recall	0.54	0.54	0.54	0.54
	Time (s)	7.2	7.4	7.9	12.4

Table 3.11: Results of learning relations over the UW-CSE dataset.

Algorithm	Metric	JMDB	Stanford	Denormalized
Aleph-FOIL	Precision	0.66	0.92	0.67
	Recall	0.44	1	0.45
	Time (m)	6.4	1,229	476.4
Aleph-Progol	Precision	0.66	1	0.69
	Recall	0.47	1	0.52
	Time (m)	312.9	1,248	937.4
Castor	Precision	1	1	1
	Recall	1	1	1
	Time (m)	15.14	108.15	32.4

Table 3.12: Results of learning relations over the IMDb dataset.

Efficiency. Besides being schema independent, Castor offers the best trade-off between effectiveness and efficiency. Generally, Aleph-FOIL is more efficient than Castor, but less effective. Aleph-Progol is usually effective, but becomes very inefficient as the size of data grows. FOIL and ProGolem only scale to small datasets.

Aleph-FOIL and Castor are the only algorithms that scale to the HIV-Large dataset. Aleph-FOIL with *clauselength* = 10 is more efficient than Castor. However, when *clauselength* is set to 15, it becomes less efficient, as shown in Table 3.10. Aleph-FOIL with both *clauselength* = 10 and 15 is also faster than Castor over the HIV-2K4K dataset. In general, top-down algorithms that follow greedy search strategies are expected to be more efficient than bottom-up algorithms. Top-down algorithms have a search bias for shorter clauses, which are cheaper to compute. They usually limit the maximum length of the clauses to be learned. Further, algorithms that follow greedy search strategies can be more efficient. Greedy algorithms are used systems that focus on efficiency [31, 67]. However, as the maximum clause length is increased, the hypothesis space grows, and these algorithms become less efficient. Top-down algorithms that do not follow a greedy search strategy, such as Progol, are generally not efficient. In our empirical studies, Aleph-Progol did not scale to the HIV-Large dataset, and is the slowest algorithm on the HIV-2K4K dataset.

Castor is able to scale to large databases such as HIV-Large and HIV-2K4K because of the optimizations explained in Section 3.6.5. By reusing information about previous coverage tests, Castor reduces the number of coverage tests on new clauses. Reusing information about coverage tests is particularly useful on large databases with complex schemas, such as the HIV datasets, where generated clauses are large and expensive to evaluate. Parallelization also helps Castor on reducing the time spent on coverage testing. For these experiments, Castor parallelized coverage testing by using 32 threads. Finally, minimization helps in reducing the size of clauses. For instance, over both of HIV datasets, Castor reduces the size of bottom-clauses over the Initial schema by 19%, over the 4NF-1 schema by 13%, and over the 4NF-2 schema by 18%, on average. Castor removes redundant literals from the bottom-clause, which results in reducing the search space and the cost of performing coverage tests. Note that the running time of all algorithms increases significantly over the 4NF-2 schema of the HIV-Large and HIV-2K4K datasets. As the *bond* relation is decomposed into *bondSource* and *bondTarget* in this schema, the number of tuples to represent bonds is doubled compared to the Initial

HIV-2K4K				
Metric	Initial	4NF-1		4NF-2
Precision	0.77	0.79		0.73
Recall	0.63	0.87		0.76
Time (m)	27	14.8		576
UW-CSE				
Metric	Original	4NF	Denorm-1	Denorm-2
Precision	0.93	0.93	0.93	0.93
Recall	0.54	0.54	0.54	0.54
Time (s)	8	8.9	9.1	13.3
IMDb				
Metric	JMDB	Stanford		Denormalized
Precision	1	0.98		1
Recall	1	0.84		1
Time (m)	7.3	90.8		8.1

Table 3.13: Results of learning over the HIV-2K4K, UW-CSE and IMDb datasets using INDs in the form of subset.

schema. Therefore, algorithms must explore clauses with a large number of literals – hundreds – whose coverage tests take a very long time. We plan to optimize the coverage-testing engine of Castor to efficiently process such datasets.

The efficiency of Castor is comparable to that of Aleph-FOIL and Aleph-Progol over the Original and 4NF schemas of the UW-CSE dataset. The running time of Aleph-FOIL and Aleph-Progol is heavily affected on the Denormalized-2 schema, as shown in Table 3.11. Castor is efficient over all schemas of this dataset. UW-CSE is the only dataset for which FOIL and ProGolem scale. However, in general, they are less efficient.

Castor is significantly more efficient and effective than Aleph-FOIL and Aleph-Progol on the IMDb dataset, as shown in Table 3.12. In general, top-down algorithms are efficient if they take the correct first steps when searching for the definition. In this case, Aleph-FOIL and Aleph-Progol (over two schemas) take the wrong steps and focus on a section of the hypothesis space that does not contain the correct definition.

General composition and decomposition. As it is explained in Section 3.6.4, there are two methods to achieve robustness over the schema variations created by the

INDs in general forms. One can use a preprocessing step to check whether the IND holds in the form of equality over the available instance. Then, one can apply the original Castor algorithm and achieve complete schema independence. The empirical results of this method are exactly the same as the ones of the original Castor algorithm with the overhead of its preprocessing step. Another method is to use the INDs in general form directly without any preprocessing. We empirically evaluate the robustness of the latter method in this section. To explore general composition and decompositions of HIV, UW-CSE, and IMDb, we restore the INDs with equality that we have enforced on their schemas to their original forms. For instance, we restore the enforced INDs with equality *movies2X[id]* = *movie[id]* in IMDb schemas to *movies2X[id]* $\subseteq$ *movie[id]* in IMDb schemas. We also modify the INDs with equality that are originally found in these datasets to INDs in form of foreign key to primary key referential integrities in their schemas. For example, we have changed INDs *movies2X[Xid]* = *X[id]* to *movies2X[Xid]* $\subseteq$ *X[id]* over IMDb schemas. Hence, the transformations explained in Section 3.8.1 for these datasets are general composition and decomposition and not bijective. We do not change the example sets. We run the extended version of Castor from Section 3.6.4 using the aforementioned INDs and all other regular INDs in each schema. For HIV-2K4K, Castor uses the INDs in Table 3.5 (bottom). For UW-CSE, Castor uses the INDs in Table 3.6 (bottom). For IMDb, Castor uses the INDs in Table 3.9 (bottom). Table 3.13 shows the results of Castor learning relations over the HIV-2K4K, UW-CSE and IMDb datasets, using only INDs in the form of subset. The extension of Castor gets the same results as in Table 3.11 over UW-CSE and is schema independent. It is also robust and delivers the same results as in Table 3.12 for JMDB and Denormalized schemas of IMDb. But, it returns precision of 0.98 and recall of 0.84 over the database with Stanford schema. Overall, it is more effective and schema independent than other algorithms over IMDb. However, the results of the extension of Castor vary with the schema over the HIV-2K4K dataset: it delivers precision of 0.77, 0.79, and 0.73 and recall of 0.63, 0.87, and 0.76 over the Initial, 4NF-1, and 4NF-2 schemas, respectively. Castor cannot access some tuples in the bottom-clause construction in these databases as explained in Section 3.6.4. Its precisions are equal or higher than the those of Aleph-FOIL and Aleph-Progol over all schemas and its recall is higher than that of Aleph-FOIL and Aleph-Progol in 4NF-2 schema. But, its recall is lower than the recall of Aleph-FOIL and Aleph-Progol over the Initial and Aleph-Progol over 4NF-1 schemas.

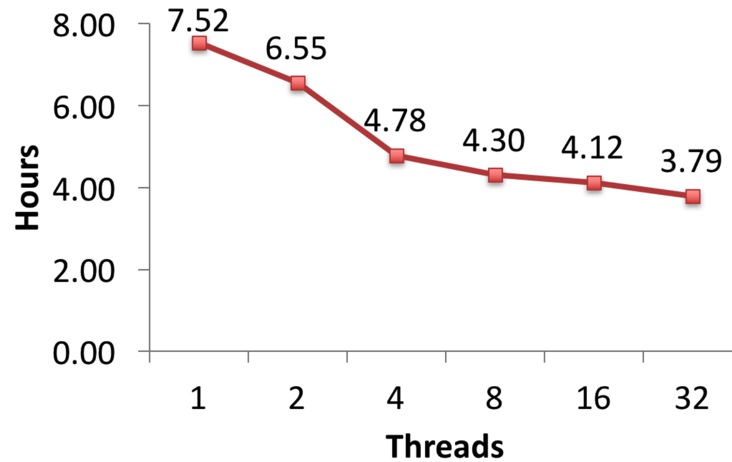

Figure 3.1: Effect of parallelization on Castor's running time over the HIV-Large dataset.

3.8.3 Effect of Castor Design Choices

We evaluate the effect of parallelization and the use of stored procedures on Castor's running time. There are some variations in the running times of Castor compared to the experiments in the previous section because we re-ran experiments, and the running times may fluctuate.

Effect of parallelization. Castor performs coverage tests in parallel to improve its running time. Figures 3.1, 3.2, and 3.3 show the effect of parallelization on Castor's running time over HIV-Large (Initial schema), HIV-2K4K (Initial schema) and IMDb (JMDB schema), respectively. Over both HIV-Large and HIV-2K4K datasets, Castor benefits from parallelization. Over the HIV-Large dataset, the best performance is obtained by using 32 threads, which reduces the running time by half compared to using 1 thread. Over the HIV-2K4K dataset, the running time also reduces significantly with parallelization and the best performance is obtained with 16 threads. Over the IMDb dataset, there is no benefit in using parallelization. In this case, Castor does not perform many coverage tests, as it is able to find the perfect definition very quickly. In this case, most of Castor's running time is spent in creating the ground bottom-clauses, as explained in Section 3.6.5. Because the UW-CSE dataset is very small, there is no need for parallelization. Notice that sequential Castor (1 thread) is more efficient than

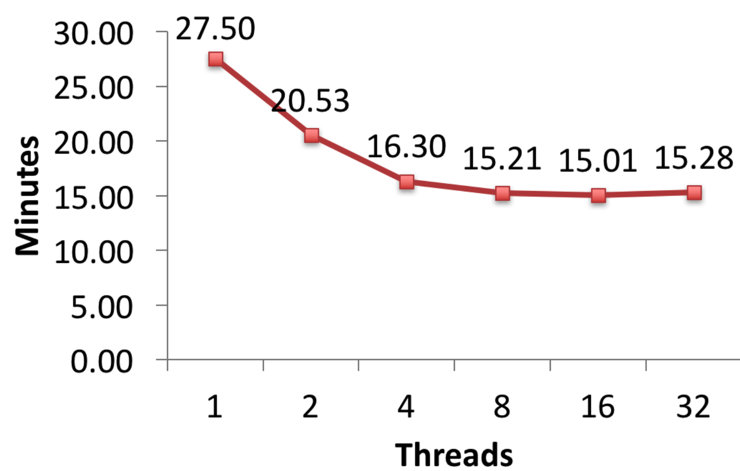

Figure 3.2: Effect of parallelization on Castor's running time over the HIV-2K4K dataset.

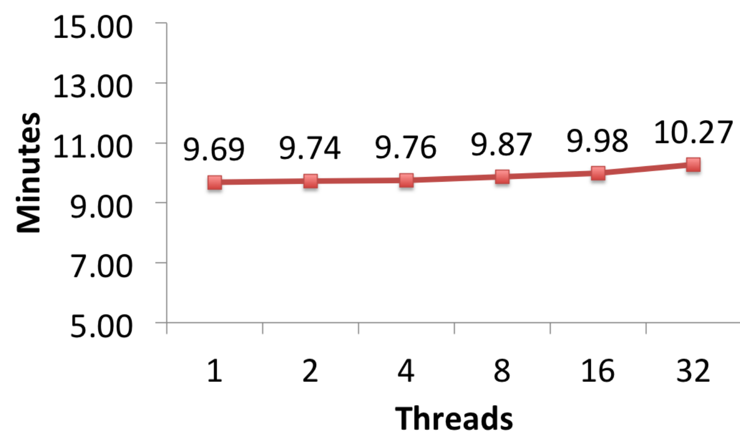

Figure 3.3: Effect of parallelization on Castor's running time over the IMDb dataset.

Dataset	With stored procedures	W/o stored procedures
HIV-Large	3.79h	4.75h
HIV-2K4K	15.28m	25.23m
IMDb	10.27m	19.49m

Table 3.14: Impact of stored procedures on Castor's running time over the HIV-Large, HIV-2K4K, and IMDb datasets.

Aleph-FOIL with *clauselength = 15* over the HIV-Large dataset and more efficient than Aleph-Progol over the HIV-2K4K and IMDb datasets. Therefore, besides parallelization, the techniques explained in Section 3.6.5 allow Castor to run efficiently.

Effect of stored procedures. Castor uses the bottom-clause construction algorithm to generate bottom-clauses in the *LearnClause* procedure, as well as to generate ground bottom-clauses, used to test coverage. As mentioned in Section 3.6.5, we implement the bottom-clause construction algorithm inside a stored procedure. To evaluate the benefit of using stored procedures, we also implement a version of Castor that does not use stored procedures. Table 3.14 shows the running time of the versions of Castor with and without stored procedures over the HIV-Large (Initial schema), HIV-2K4K (Initial schema) and IMDb (JMDB schema) datasets. The version of Castor that uses stored procedures obtains between 1.25x and 1.9x speedup over the version that does not use stored procedures.

3.8.4 Query-based Algorithms

We used the interactive algorithm with automatic user mode in the LogAn-H system. In this mode, the system is told the Horn definition to be learned, so that it can act as an oracle. Then the algorithm's queries are answered automatically until it learns the exact definition. When answering EQs, the counter examples are produced by the system. Therefore, LogAn-H only takes as input the schema of the dataset, but not the database instance. We performed experiments using the schemas of the UW-CSE dataset. We generated random Horn definitions over the Denormalized-2 schema of the UW-CSE dataset. The definition generator has a parameter to indicate the number of variables in each clause. To generate the head of each clause, we created a new relation

of random arity, where the minimum arity is 1 and the maximum arity is the maximum arity of the relations in the Denormalized-2 schema. The body of each clause can be of any length as long as the number of variables in the clause is equal to the specified parameter and all variables appearing in the head relation also appear in any relation in the body. The body of the clause is composed of randomly chosen relations, where each relation can be the head relation or any relation in the input schema. Head and body relations are populated with variables, where each variable is randomly chosen to be an existing or new variable.

After generating each random Horn definitions over the Denormalized-2 schema, we transformed these expressions to the Denormalized-1, 4NF and Original schemas by simply doing vertical decomposition to each of the clauses in a definition. We varied the number of clauses in a definition to be between 1 and 5, each containing between 4 and 8 variables. Therefore, we generated 50 random definitions for each setting. We ran the LogAn-H system and recorded the number of queries required to learn each definition under each schema. The number of EQs and MQs asked by the algorithm are presented in Figure 3.4 and 3.5, respectively. The average number of EQs required by the A2 algorithm is constant for different number of variables and similar throughout all schemas. However, the average number of MQs required by the A2 algorithm varies with the schema. Particularly, the number of MQs is greater for more decomposed schemas, e.g., the Original schema. Further, the number of MQs also increases with the number of variables. This difference of MQs between the schemas originates from a step in the A2 algorithm that removes non-essential literals in ground bottom-clauses generated from negative examples. This process is similar to Castor's negative reduction. It removes a literal and asks an MQ to verify whether the example is still negative. Therefore, the more decomposed the schema is, the more literals can be removed, hence more MQs are asked.

3.9 Related Work

There has been a growing interest in developing relational learning algorithms that scale to large databases [16, 31, 67]. AMIE [31] and Ontological Pathfinding [16] focus on learning first-order rules from RDF-style knowledge bases. They impose several restrictions on the learned rules to be able to learn over large knowledge bases. QuickFOIL [67]

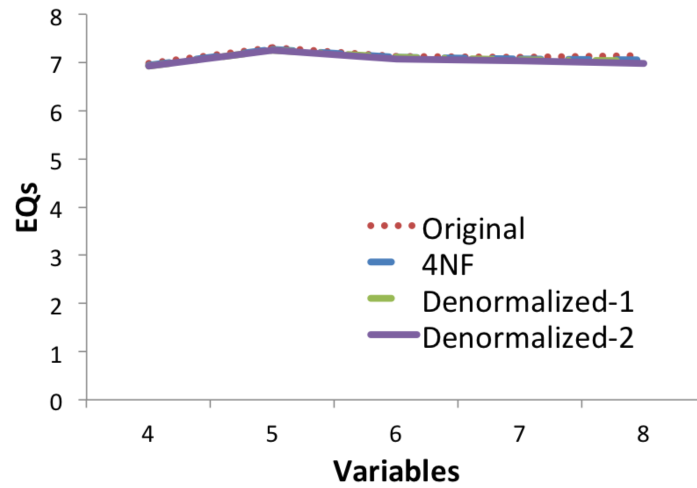

Figure 3.4: Average number of equivalence queries by the A2 algorithm.

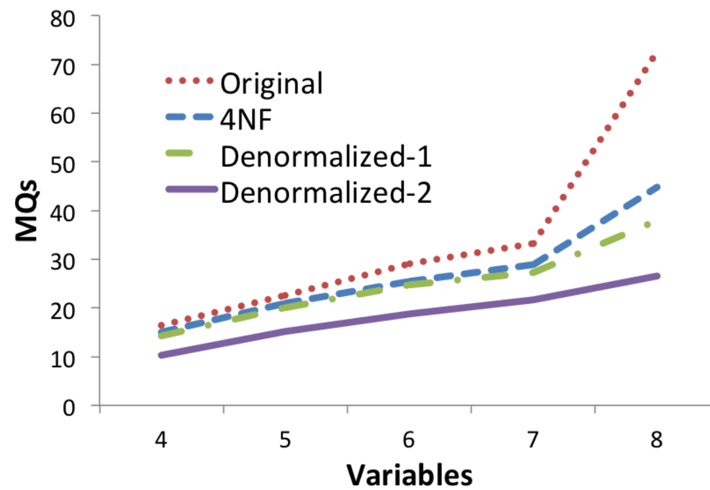

Figure 3.5: Average number of membership queries by the A2 algorithm.

provides an in-RDBMS implementation of a modified version of FOIL. Besides efficiency, we also focus on schema independence.

Davis et al. [21] use a relational learning algorithm to learn new features, which are used to improve the performance of an Statistical Relational Learning (SRL) model. The proposed system in this chapter can also be used to learn these features as well as the structure of the SRL models, with the added benefit of efficiency and schema independence.

We build upon the body of work on transforming databases without modifying their content by exploring the sensitivity of relational learning algorithms to such transformations [29, 40]. Another notable group of database transformations is schema mapping for data exchange [28]. These transformations generally lose information and introduce incomplete information to a database. However, for the property of schema independence, a transformation should preserve the information content of databases. Fagin explores invertible schema mappings that preserve the information content of database instances [27]. Nevertheless, these mappings may introduce labeled nulls to the database instance. Similar to answering queries over databases with labeled nulls, we believe that it is challenging to define reasonable semantics and design efficient and effective algorithms for learning relations over such databases. Researchers have defined the property of design independence for keyword query processing over XML [63]. We extend this line of work by exploring the schema independence for relational learning algorithms.

The architects of the relational model have argued for logical data independence, which oversimplifying a bit, means that an exact query should return the same answers no matter which logical schema is chosen for the data [2, 18]. In this chapter, we extend the principle of logical data independence [2] for relational learning algorithms. The property of schema independence also differs with the idea of logical data independence in a subtle but important issue. One may achieve logical data independence by an affordable amount of experts' interventions, e.g., defining views over the database. However, it takes deeper expertise to find the proper schema for a learning algorithm, particularly for database applications that contain more than a single learning algorithm. Hence, it is less likely to achieve schema independence via experts' interventions.

Chapter 4: Automatically Setting Language Bias

4.1 Motivation

The space of possible hypotheses that a relational learning algorithm can explore consists of all Datalog programs defined over the schema of the input database. This space can be very large if the schema of the input database contains many relations or many attributes. Therefore, users must constraint the hypothesis space of relational learning algorithms using a *language bias*. One form of language bias is *syntactic bias*, which restricts the structure and syntax of the learned Datalog programs. Relational learning systems usually allow users to specify the syntactic bias through statements called *predicate definitions* and *mode definitions* [22]. Predicate and mode definitions express several types of restrictions on the structure of the learned Datalog programs, such as the relations allowed to be in the Datalog program, whether an attribute can appear as a variable or constant, and whether two relations can join. Consider the UW-CSE database (*alchemy.cs.washington.edu/data/uw-cse*), which contains information about a computer science department and whose schema is shown in Table 4.1. Table 4.2 shows a fragment of predicate and mode definitions used for the UW-CSE database. A detailed explanation of these definitions is given in Section 4.2. To the best of our knowledge, all (statistical) relational learning systems require some form of syntactic bias to restrict the hypothesis space. Malec et al. [45] have shown that predicate and mode definitions significantly reduce the running time of (statistical) relational learning algorithms.

For a relational learning algorithm to be effective and efficient, predicate and mode definitions must encode a great deal of information about the structure of the learned

student(stud)	professor(prof)
inPhase(stud, phase)	hasPosition(prof, position)
yearsInProgram(stud, years)	taughtBy(course, prof, term)
courseLevel(course, level)	ta(course, stud, term)
publication(title, person)	

Table 4.1: Schema for the UW-CSE dataset.

Predicate definitions	Mode definitions
student(T1)	advisedBy(+,+)
inPhase(T1,T2)	student(+)
professor(T3)	inPhase(+,-)
hasPosition(T3,T4)	inPhase(+,#)
publication(T5,T1)	professor(+)
publication(T5,T3)	hasPosition(+,-)
...	

Table 4.2: A subset of predicate and mode definitions for the UW-CSE dataset.

Datalog programs [22]. A user should both know the internals of the learning algorithm and the schema of the input database and have a relatively clear intuition on the structure of effective Datalog programs for the target relation to set a sufficient degree of restriction. However, there may not be any user that both knows the database concepts, such as schema, and has a clear intuition about the target relation. Furthermore, the number of predicate and mode definitions of is generally large and hard to debug and maintain. Users normally improve the initial set of definitions via trial and error, which is a tedious and time-consuming process. Hence, it takes a lot of time and effort to write and maintain these definitions, particularly for a relatively complex schema. In our conversations with (statistical) relational learning experts, they have called predicate and mode definitions the "black magic" needed to make relational learning work and believe them to be a major reason for the relative unpopularity of these algorithms among users.

We propose AutoMode, a system that leverages the information in the schema and content of the database to automatically generate predicate and mode definitions. The predicate and mode definitions generated by AutoMode can be used by relational learning systems such as Castor (Chapter 3). We show that the predicate and mode definitions produced by AutoMode deliver the same accuracy as the manually written and tuned ones while imposing only a modest running-time overhead over large real-world databases.

Relational learning algorithms do not generally scale to large databases. Generating the language bias automatically through AutoMode may result in an under-restricted hypothesis space. Therefore, with such a language bias, it may be extremely time-consuming to learn over large databases. We propose to use sampling techniques to get a subset of the data that is used to generate candidate definitions. We study different

sampling techniques and integrate them into the algorithm that builds candidate definitions in Castor. We show that the effectiveness and efficiency of the learning algorithms can improve with the appropriate sampling techniques.

4.2 Language Bias

In relational learning algorithms, language bias restricts the structure and syntax of the generated clauses. Language bias is specified through predicate and mode definitions [22].

Predicate definitions assign one or more *types* to each attribute in a database relation. In a candidate clause, two relations can be joined over two attributes (i.e., attributes are assigned the same variable) only if the attributes have the same type. For instance, in Table 4.2, the predicate definition `student(T1)` indicates that the attribute in relation *student* is of type `T1`, and the predicate definition `inPhase(T1,T2)` indicates that the first and second attributes of relation *inPhase* are of type `T1` and `T2`, respectively. Therefore, relations *student* and *inPhase* can be joined on attributes *student[stud]* and *inPhase[stud]*. It is possible to assign multiple types to an attribute. For example the predicate definitions `publication(T5,T1)` and `publication(T5,T3)` indicate that the attribute *author* in relation *publication* belongs to both types `T1` and `T3`. Predicate definitions restrict the joins that can appear in a candidate clause: two relations can be joined only if their attributes share a type.

Mode definitions indicate whether a term in an atom should be a new variable, i.e., existentially quantified variable, an existing variable, i.e., appears in a previously added atom, or a constant. They do so by assigning one or more symbols to each attribute in a relation. Symbol $+$ indicates that a term must be an existing variable, except for the atom in the head of a Horn clause. Symbol $-$ indicates that a term can be an existing variable or a new variable. For instance, the mode definition `inPhase(+,-)` in Table 4.2 indicates that the first term must be an existing variable and the second term can be either an existing or a new variable. Symbol # indicates that a term should be a constant. For instance, the mode definition `inPhase(+,#)` indicates that the second term must be a constant. Each atom in a candidate clause must satisfy at least one mode definition.

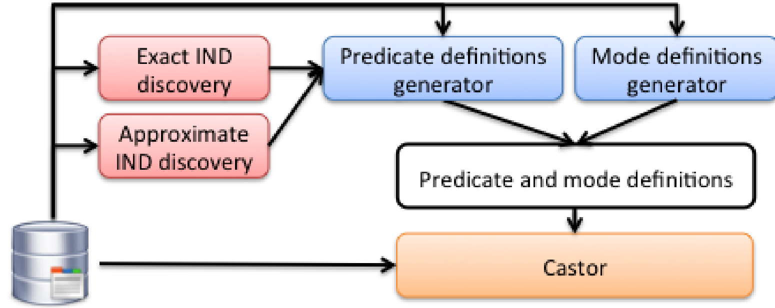

Figure 4.1: Architecture of the AutoMode system.

4.3 AutoMode System

We propose AutoMode, a system that leverages the information in the schema and content of the database to automatically generate predicate and mode definitions. Figure 4.1 shows the components of the AutoMode system. The predicate and mode definitions generated by AutoMode can be used by relational learning systems such as Castor (Chapter 3). AutoMode reads and extracts the information about the schema of the underlying database from the relational database management system (RDBMS). It then generates predicate and mode definitions in a pre-processing step. Castor uses these definitions to learn the definition of some target relation. The same predicate and mode definitions can be used to learn different target relations.

4.3.1 Generating Predicate Definitions

Let R and S be two relation symbols in the schema of the underlying database. Let $R(e_1, \cdots, e_n)$ and $S(o_1, \cdots, o_m)$ be two atoms in a clause C. Let e_i be the term in attribute $R[A]$ and o_j be the term in attribute $S[B]$, and let e_i and o_j be assigned the same variable or constant. That is, clause C joins R and S on A and B. Clause C is satisfiable only if these attributes share some values in the input database. Typically, the more frequently used joins are the ones over the attributes that participate in inclusion dependencies (INDs), such as foreign-key to primary-key referential constraints. AutoMode uses INDs in the input database to find which attributes, among all relations, share the same type. Let X and Y be sets of attribute names in R and S, respectively. Let I_R and

I_S be the relations of R and S in the database. Relations I_R and I_S satisfy *exact IND* (*IND* for short) $R[X] \subseteq S[Y]$ if $\pi_X(I_R) \subseteq \pi_Y(I_S)$. If X and Y each contain only a single attribute, the IND is a *unary IND*. Given IND $R[X] \subseteq S[Y]$ in a database, the database satisfies unary IND $R[A] \subseteq S[B]$, where $A \in X$ and $B \in Y$. INDs are normally stored in the schema of the database. If they are not available in the schema, one can extract them from the database content. AutoMode uses the Binder algorithm [52] to discover INDs from the database, shown by the **Exact IND discovery** box in Figure 4.1, and generates all unary INDs implied by them.

We have observed that using exact INDs is not enough for generating helpful predicate definitions. For instance, consider using the UW-CSE database, whose schema fragments are shown in Table 4.1. Consider the task of learning a definition for the relation *advisedBy(stud, prof)*, which indicates that the student *stud* is advised by professor *prof*. A relational learning algorithm may learn the following Datalog program for the *advisedBy* relation:

$$\mathit{advisedBy}(x, y) \leftarrow \mathit{student}(x), \mathit{professor}(y), \mathit{publication}(z, x), \mathit{publication}(z, y)$$

which indicates that a student is advised by a professor if they have been co-authors of a publication. This definition requires joining relations *publication*, *student*, and *professor* on attributes *publication[author]*, *student[stud]*, and *professor[prof]*. However, the UW-CSE database does not satisfy INDs *publication[author]* $\subseteq$ *student[stud]* or *publication[author]* $\subseteq$ *professor[prof]* because *publication[author]* contains both students and professors. Hence, AutoMode also uses *approximate INDs* to assign types to attributes. In an *approximate unary IND* $(R[A] \subseteq S[B], \alpha)$, one has to remove at least α fraction of the distinct values in $R[A]$ so that the database satisfies $R[A] \subseteq S[B]$ [1]. Approximate INDs are not usually maintained in a schema and are instead discovered from the database content. We have implemented a program to extract approximate INDs from the database, shown by the **Approximate IND discovery** box in Figure 4.1. We use a relatively high error rate, 50%, for the approximate INDs to allow for a flexible hypothesis space.

After discovering unary exact and approximate INDs, AutoMode runs Algorithm 7 to generate a directed graph called *type graph*, which it then uses to assign types to attributes. First, it creates a graph whose nodes are attributes in the input schema

and has an edge between each pair of attributes that participate in an exact or approximate IND. Figure 4.2 shows an example of the type graph containing a subset of the attributes in the UW-CSE schema, where edges corresponding to exact and approximate INDs are shown by solid and dashed lines, respectively. If there are both approximate INDs $(R[A] \subseteq S[B], \alpha_1)$ and $(S[B] \subseteq R[A], \alpha_2)$, AutoMode uses only the one with lower error rate. The algorithm then assigns a new type to every node in the graph without any outgoing edges. For example, it assigns new types `T1`, `T3`, and `T5` to *student[stud]*, *professor[prof]*, and *publication[title]*, respectively, in Figure 4.2. If there are cycles in the type graph, the algorithm assigns the same new type to all nodes in each cycle. Next, it propagates the assigned type of each attribute to its neighbors in the reverse direction of edges in the graph until no changes are made to the graph. For example, in Figure 4.2, the algorithm propagates type `T1` to *inPhase[stud]* and *ta[stud]* and attribute *publication[author]* inherits types `T1` and `T3` from *student[stud]* and *professor[prof]*, respectively. Because the error rates of approximate INDs accumulate over multiple edges in the graph, AutoMode propagates types only once over edges that correspond to approximate INDs.

Algorithm 7: Algorithm to generate the type graph.

Input : Schema $\mathcal{S}$ and all unary INDs Σ.
Output: Type graph G.
create graph $G = (V, E)$ where V contains a node for each attribute in the schema and $E = \varnothing$
foreach *IND* $R[A] \subseteq S[B] \in \Sigma$ **do**
  add edge $v \rightarrow u$ to E, where v and u correspond to attributes $R[A]$ and $S[B]$, respectively
foreach *node* $u \in V$ *without outgoing edges* **do**
  generate new type T and set $types(u) = \{T\}$
foreach *cycle* $K \subseteq V$ **do**
  generate new type T and set $types(u) = \{T\}\ \forall u \in K$
repeat
  foreach $v \rightarrow u \in E$ *where* $types(u) \neq \varnothing$ **do**
    set $types(v) = types(v) \cup types(u)$
until *no changes in* G
return G

Given the resulting graph, for each relation, AutoMode computes the Cartesian prod-

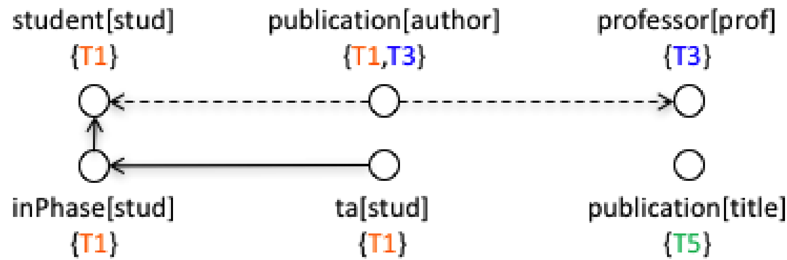

Figure 4.2: A fragment of the type graph for the UW-CSE dataset. Solid lines represent exact INDs and dashed lines represent approximate INDs.

uct of the types associated with its attributes. For each tuple in this Cartesian product, it produces a predicate definition for the relation. For instance, given the type assignment in Figure 4.2, AutoMode generates predicate definitions `publication(T5,T1)` and `publication(T5,T3)` for the *publication* relation.

4.3.2 Generating Mode Definitions

AutoMode allows every attribute of every relation be a variable. However, it forces at least one variable in an atom to be an existing variable, i.e., appears in previously added atoms, to avoid generating Cartesian products in the clause. For each attribute A in relation R, AutoMode generates a mode definition for R where attribute A is assigned the $+$ symbol and all other attributes are assigned the $-$ symbol. Hence, all attributes are allowed to have new variables except the attribute with symbol $+$. For instance, AutoMode generates the mode definitions `publication(+,-)` and `publication(-,+)` for relation *publication* in Table 4.1.

AutoMode uses a hyper-parameter called *constant-threshold* to determine whether an attribute can be a constant. The value for constant-threshold can take an absolute or a relative threshold. If it is an absolute threshold, AutoMode allows an attribute to be a constant if the number of distinct values in the attribute is below the value of constant-threshold. If it is a relative threshold, AutoMode allows an attribute to be a constant if the ratio of distinct values of the attribute to the total number of tuples in the relation is below the value of constant-threshold. This hyper-parameter has a relatively intuitive meaning.

For each relation R in the database, AutoMode finds all attributes in R that can be constants using the aforementioned rule. Then, it computes the power set $\mathbf{M}$ of these attributes. For each non-empty set $M \in \mathbf{M}$, AutoMode generates a new set of mode definitions where it assigns $+$ and $-$ symbols as described above, except for the attributes in M, which are assigned the # symbol. For example, AutoMode finds that the number of values in attribute *phase* of relation *inPhase* in Table 4.1 is smaller than the input threshold. Then, this attribute can be constant and AutoMode generates the mode definition `inPhase(+,#)` for relation *inPhase*.

4.4 Improving Efficiency Through Sampling

When learning over large databases, relational learning algorithms can be inefficient. Further, if the language bias written by an expert or generated by AutoMode is not restrictive enough, algorithms can be extremely inefficient. In this section, we study sampling techniques that allow relational learning algorithms to learn efficiently. We implement these techniques in Castor (Chapter 3). However, the same techniques can be implemented in other algorithms that build bottom-clauses, such as Progol [47] and ProGolem [50].

A *bottom-clause* C_e associated with an example e is the most specific clause in the hypothesis space that covers e. The bottom-clause construction algorithm consists of two phases (refer to Section 3.5.1). First, it finds all the information in I relevant to e, denoted by I_e. Then, given the information relevant to e, it creates the bottom-clause C_e. The tuple set I_e may be large if many tuples in I are relevant to e. Thus, bottom-clause C_e would be very large, making the learning process prohibitively expensive. To overcome this problem, it is necessary to obtain a smaller tuple set $I_e^s \subseteq I_e$. Then, the bottom-clause C_e is created based on the tuples in I_e^s, instead of I_e. Existing algorithms, such as Progol [47] and ProGolem [50], already use sampling to build bottom-clauses. They use a technique that we call naïve sampling, which we explain below. We propose two other sampling techniques called random sampling and stratified sampling. We implement the three sampling techniques in Castor and empirically evaluate them in Section 4.5.2.

4.4.1 Naïve Sampling

Let C_e be a bottom-clause associated with example e. A *naïve sample* C_e^s of clause C_e is the clause obtained the following way. Let I_R be the set of tuples in relation R that can be added to I_e. The naïve sampling algorithm obtains a *uniform* sample I_R^s of I_R and only adds tuples in I_R^s to I_e. In a uniform sample, every tuple in I_R is sampled independently with the same inclusion probability, i.e., $\forall t \in I_R, p(t) = \frac{1}{|I_R|}$.

4.4.2 Random Sampling

Let *sample*(I) be a random sample of I. One way to obtain a smaller tuple set $I_e^s \subseteq I_e$ is to obtain a random sample $I_e^s = sample(I_e)$. Let the *inclusion probability* $p(t)$ of tuple $t \in I_e$ be the probability that t is included in *sample*(I_e). The inclusion probability of t should be proportional to the number of tuples that are connected to e through t. Let t and t' be two tuples in I_e and let T and T' be the sets of tuples connected to e through t and t', respectively. If $|T| > |T'|$, then the inclusion probability $p(t)$ of t should be greater than the inclusion probability $p(t')$ of t'. This idea is similar to the idea of performing random sampling over joins [15].

A *join tree* is a tree structure where nodes represent relations or tuples of a relation. Let n_R represent a node in the join tree that represents relation R. A node n_{R_1} in the join tree has a child n_{R_2} if R_1 can join with R_2. Let T be the target relation and $e = T(t)$ be a training example. We create a join tree G where the root is e and the children of each node n_R in the tree are nodes representing all relations in the schema that can join with R. The depth of the tree is limited by the given parameter d. Let $R \prec R_i$ denote that n_R is an ancestor of n_{R_i} in the join tree G. For any relation R and any tuple $t \in I_R$, let the weight of t be $w(t) = |t \bowtie (\bowtie_{i:R \prec R_i} R_i)|$ and $w(R) = \Sigma_{t \in I_R} w(t)$. In random sampling, the inclusion probability of a tuple $t \in I_R$ is $p(t) = \frac{w(t)}{w(R)}$.

The bottom-clause construction algorithm using random sampling is depicted in Algorithm 8. Computing the weight $w(t)$ requires computing all join paths that can be derived from t. To sample a tuple from I_R, we would have to compute $w(t)$ for all $t \in I_R$. Instead, we use the sampling technique proposed by Olken [51]. We use Algorithm 8, but we change the way that we sample tuples. The algorithm samples a tuple $t \in (T \rtimes R)$ uniformly, i.e., with probability $\frac{1}{|T \rtimes R|}$. Let $I_R = (T \rtimes R)$. Then, the algorithm accepts

or rejects tuple t with a probability based on the frequency of value $v = t[A]$ on I_R, denoted as $d_A(v, I_R) = |t : t \in I_R, t[A] = v|$. Let $D_A(I_R) = \max_v d_A(v, I_R)$. That is, tuple t is accepted with probability $\frac{d_A(v,I_R)}{D_A(I_R)}$ and rejected otherwise. If accepted, tuple t is added to S and T_R. Otherwise, the algorithm samples again until s tuples have been accepted.

Algorithm 8: Bottom-clause construction algorithm using random sampling.

```
Input  : example e, # of iterations d, sample size s
Output: Bottom-clause C
I_e^s = {}
foreach child relation R_i of R do
    I_e^s = I_e^s ∪ Acyclic-Sample({t}, R_i, 1, d, s)
C = create bottom-clause from I_e^s
return C
Function Acyclic-Sample(T, R, i, d, s):
    S = {}, T_R = {}
    for i = 1 ... s do
        t ← a random tuple t ∈ (T ⋊ R) with probability w(t)/w(T ⋊ R)
        S = S ∪ {t}
        T_R = T_R ∪ {t}
    if i < d then
        foreach child relation R_j of R do
            S = S ∪ Acyclic-Sample(T_R, R_j, i + 1, d, s)
    return S
```

4.4.3 Stratified Sampling

Let G be a join tree with root node set to example e. Let S be a relation that contains attribute A, where A can appear as a constant according to the language bias. We extend the join tree G to contain a new node for each distinct value in $S[A]$. The parents of these nodes are the same as n_S, the node for relation S.

Given a node n_R in G, we define a *stratum* for each child of n_R. Therefore, there is a stratum for each relation S that can join with R and, if S contains an attribute A that can be a constant, there is a stratum for each each distinct value in $S[A]$. A *stratified sample* I_e^s of I_e is a subset of I_e that contains at least one tuple for each stratum in

G. A stratified sample C_e^s of clause C_e is the clause created from the stratified sample I_e^s of I_e. Algorithm 9 depicts the bottom-clause construction algorithm using stratified sampling. The algorithm traverses join tree G in a depth-first manner. Once it reaches a given depth, it computes the strata in the current relation, e.g., relation S, and samples a number of tuples for each stratum in S and adds them to I_e. When the algorithm backtracks to the parent relation R of S, it adds all tuples in R that join with the sampled tuples in S to I_e.

Algorithm 9: Bottom-clause construction algorithm using stratified sampling.

```
Input  : example e, # of iterations d, sample size s
Output: bottom-clause C_e
I_e^s = {}
foreach attribute A in e do
    foreach relation R containing attribute A do
        I_e^s = I_e^s ∪ StratRec(R, A, {e[A]}, 1, d, s)
C_e = create clause from e and I_e^s
return C_e
Function StratRec(R, A, M, i, d, s):
    I_e^s = {}
    I_R = σ_{A∈M}(R)
    if i = d (last iteration) then
        I_e^s = I_e^s ∪ SampleStrata(I_R, s)
    else
        foreach attribute B in R do
            foreach relation S containing attribute B do
                I_S = StratRec(S, B, π_B(I_R), i + 1, d, s)
                I_e^s = I_e^s ∪ (σ_{B∈π_B(I_S)}(I_R))
    return I_e^s
```

4.5 Empirical Results

We empirically evaluate AutoMode and the proposed sampling techniques to answer the following questions:

1. What is the benefit of setting a language bias in relational learning? (Section 4.5.1)
2. How does Castor perform when using language bias generated by AutoMode com-

Name	#R	#T	#P	#N
UW-CSE	9	1.8K	102	204
HIV	80	14M	2K	4K
IMDb	46	8.4M	1.8K	3.6K

Table 4.3: Number of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset.

pared to using language bias written by an expert? (Section 4.5.1)

3. How do sampling techniques affect the efficiency and effectiveness of Castor? (Section 4.5.2)

4.5.1 Evaluating AutoMode

We run experiments over three datasets whose information is shown in Table 4.3. The UW-CSE database contains information about a computer science department. We learn the target relation *advisedBy(stud, prof)*, which indicates that student *stud* is advised by professor *prof*. The HIV database contains structural information about chemical compounds (*wiki.nci.nih.gov/display/NCIDTPdata*). We learn the target relation *anti-HIV(comp)*, which indicates that compound with id *comp* has anti-HIV activity. In this dataset, the positive examples are compounds known to have anti-HIV activity, while negative examples are compounds known to lack anti-HIV activity. The IMDb database (*imdb.com*) contains contains information about movies and people who make them. We learn the target relation *dramaDirector(dir)*, which indicates that person with id *dir* has directed a drama movie. Over the UW-CSE and IMDb databases, we generate the negative examples by using the closed-world assumption, and then sample to obtain twice as many negative examples as positive examples.

We use four methods of setting language bias.

1. **Baseline** assigns the same types to all attributes and allows every attribute to be a variable or a constant.

2. **Baseline without constants** is the same as the baseline method, except that it does not allow any attribute to be a constant.

3. **Manual tuning** uses the language bias written by an expert who has knowledge of the relational learning system and knows how to write predicate and mode definitions. The author of this thesis acted as the expert. The expert had to learn the schema and go through several trial and error phases by running the underlying learning system and observing its results to write the predicate and mode definitions. The expert has written 36, 165, and 112 predicate and mode definitions for the UW-CSE, HIV, and IMDb databases, respectively.

4. **AutoMode** uses the automatically generated predicate and mode definitions, as described in Section 4.3. The original databases do not contain INDs. Therefore, AutoMode calls the IND discovery tools shown in Figure 4.1. The pre-processing step of AutoMode to extract INDs takes 2 seconds, 45 minutes, and 53 minutes over the UW-CSE, HIV, and IMDb databases, respectively. We set the hyper-parameter in AutoMode used to determine whether an attribute can be a constant (Section 4.3.2) to 5 for UW-CSE, 20 for HIV, and 400 for IMDb.

We compare the quality of the learned definitions using the metrics of *precision* and *recall* [22]. We also compare the learning time of Castor to show the effect of predicate and mode definitions. We perform 10-fold cross validation for HIV and IMDb datasets and 5-fold cross validation for UW-CSE. We evaluate precision, recall, and learning time, showing the average over the cross validation. We run experiments on a 2.3GHz Intel Xeon E5-2670 processor, running CentOS Linux 7.2 with 500GB of main memory.

The results are shown in Table 4.4. We analyze the results of each setting.

Baseline. Over the UW-CSE database, Castor is less accurate and efficient compared to other settings. Over the HIV database, Castor does not terminate after 36 hours. Over the IMDb database, Castor is killed by the kernel because of extreme use of resources. By allowing every attribute to be a constant, every value in the database – even if it has a non-predictive value – may appear in a literal as a constant. Therefore, the generated bottom-clause contains a large number of literals, many of which are not useful for learning a definition for the target relation. For instance, the first bottom-clause created when running over the IMDb databases contains on average 1255 literals. Further, by assigning the same type to all attributes in all relations, it allows all relations to join with each other on any attribute, resulting in a long running time.

Baseline without constants. Over the UW-CSE database, this setting is the most

efficient, and obtains competitive precision and recall compared to manual tuning and AutoMode. Over the HIV database, Castor is able to learn a reasonable definition, but less accurate than manual tuning and AutoMode. However, because this setting uses the same type for all attributes, it allows all relations to join with each other over any attribute, resulting in a long running time. Over the IMDb database, the perfect definition for the target relation *dramaDirector* contains a constant. However, in this setting, constants are not allowed. Therefore, Castor learns other definitions which are significantly less accurate compared to manual tuning or AutoMode.

Manual tuning. Castor obtains similar precision and recall using manual tuning and AutoMode. Castor with manual tuning is very efficient. However, an expert had to spend significant amount of time tuning the language bias. Further, a non-expert user would not be able to specify this bias.

AutoMode. AutoMode is more effective than the baselines, and as effective as manual tuning. However, AutoMode is slightly less efficient than manual tuning. Manually written predicate and mode definitions provide a more restricted hypothesis space than the ones generated by AutoMode. Thus, Castor has to explore a larger hypothesis space when using AutoMode. Nevertheless, the overhead in the running time is about 18 minutes for the HIV database and 4 minutes for the IMDb database, which is a reasonable overhead for saving an expert's time and the enterprise's financial resources that pay the machine learning expert. There is no overhead over the UW-CSE database. Hence, we argue that automating the generation of predicate and mode definitions with the cost of a modest overhead in performance is a reasonable trade-off. Further, AutoMode enables non-experts to use relational learning systems more easily.

4.5.2 Evaluating Sampling Techniques

In this section, we empirically evaluate the sampling techniques presented in Section 4.4. We use two versions of the HIV dataset. Both datasets contain the same background knowledge, but we vary the number of training examples. The first version, which we call HIV-Large, contains 5.8K positive and 36.8K negative examples. The second version, which we call HIV-2K4K, contains 2K positive and 4K negative examples. The schema of the HIV dataset used in Section 4.5.1 contains one relation for each type of element and each bond type. We modify the schema so that the types of element and bonds are

Dataset	Measure	Baseline	Baseline (w/o const.)	Manual tuning	AutoMode
UW-CSE	Precision	0.76	0.96	0.93	0.93
	Recall	0.50	0.48	0.54	0.54
	Time	47s	6.6s	11s	10.8s
HIV	Precision	-	0.72	0.77	0.77
	Recall	-	0.91	0.89	0.89
	Time	>36h	20h	14.7m	32.2m
IMDb	Precision	-	0.68	1	1
	Recall	-	0.51	1	1
	Time	-	9.2h	2.7m	6.9m

Table 4.4: Results of learning relations over UW-CSE, HIV, and IMDb data (h=hours, m=minutes, s=seconds).

compounds(compound, atom)	atoms(atom, element)
atoms_p2(atom, type)	atoms_p3(atom, type)
bonds(bond, atom1, atom2, type)	

Table 4.5: New schema for the HIV dataset.

stored in an attribute. The new schema is shown in Table 4.5. This schema follows a more common database design. Further, it allows a better comparison of the sampling techniques.

We use three versions of Castor. Each version uses a different sampling technique for bottom-clause construction:

1. **Castor-Naïve** uses naïve sampling, as explained in Section 4.4.1.

2. **Castor-Random** uses random sampling, as explained in Section 4.4.2. Random sampling is implemented using Olken's sampling approach.

3. **Castor-Stratified** uses stratified sampling, as explained in Section 4.4.3.

Table 4.6 shows the results of Castor learning over the HIV-Large and HIV-2K4K datasets with different sampling techniques. Over the HIV-Large dataset, Castor-Stratified obtains the best precision and recall and is the most efficient. Castor-Random obtains a better precision than Castor-Naïve, but lower recall. Over the HIV-2K4K dataset, Castor-Stratified obtains the best recall and a competitive precision. Castor-Random

Dataset	Measure	Castor-Naïve	Castor-Random	Castor-Stratified
HIV-Large	Precision	0.80	0.84	0.87
	Recall	0.77	0.73	0.88
	Time	7.4h	6.3h	6.1h
HIV-2K4K	Precision	0.74	0.83	0.81
	Recall	0.84	0.86	0.91
	Time	22.3m	20.2m	24.3m

Table 4.6: Results of learning relations over HIV data with different sampling techniques (h=hours, m=minutes).

obtains the best precision and a competitive recall and is the most efficient. Castor-Naïve obtains the lowest precision and recall. In the HIV background knowledge, shared by both datasets, compounds contain hundreds of atoms. Some atoms are common elements, e.g., Hydrogen, while other atoms are rare elements, e.g., Lithium. Castor-Stratified is able to explore join paths that lead to all types of elements in a compound. Therefore, the bottom-clauses generated by Castor-Stratified contain diverse information, which allows it to learn better definitions. Castor-Random focuses on obtaining a subset of the data that is representative of the whole data, but is not necessarily helpful for learning definitions. Castor-Naïve simply samples tuples uniformly in each relation.

Chapter 5: Robustness Against Content Heterogeneities

5.1 Motivation

Users often use machine learning methods to discover interesting and novel relations over relational databases [22, 34, 67]. For instance, consider the IMDb database (*imdb.com*), which contains information about movies, for which schema fragments are shown in Table 5.1 (top). Given this database, a user may want to use a relational learning algorithm such as Castor to find the definition for the new relation *highGrossing(title)*, which indicates that the movie with a given title is high grossing. 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 the following definition:

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

which indicates that high grossing movies are often released in May and their genre is *comedy*.

Databases often contain heterogeneities in representing data values [12, 24, 30, 33], which may prevent the 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 budget or total grossing of movies. This information is available in another database called Box Office Mojo

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

Table 5.1: Schema fragments for the IMDb and Box Office Mojo datasets.

(BOM) (*boxofficemojo.com*), for which schema fragments are shown in Table 5.1 (bottom). Thus, to learn an accurate definition for the *highGrossing* relation, the user has to collect (more) data from the BOM database. For instance, the information about the budget of movies may be helpful in learning an effective definition for the *highGrossing* relation as high grossing movies may have high budgets. Thus, users may integrate these two databases under a unified schema over which to learn the target relation. However, the same entity or value may be represented in various forms in the integrated database. For instance, 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*. In this case, the learning algorithm *cannot* learn the aforementioned accurate definition for the *highGrossing* relation as it will *not* be able to connect the information about the same movie from IMDb and BOM movie relations. Such heterogeneities in representing names of entities and data values may also appear in a single data source due to the inconsistencies in data entry and collection [12, 30].

Currently, users must manually resolve the heterogeneities in representing data values and then learn over the clean database. However, it is difficult, time-consuming, and labor-intensive to resolve these heterogeneities over large databases [24, 33]. First, the user has to develop and train a matching function that distinguishes and unifies different values that refer to the same entity, apply that function on the database, and materialize 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 to produce and materialize a clean database instance. Third, the process of unifying values may produce numerous clean database instances as one value may match multiple distinct values [12, 13, 30]. For example, title *Star Wars* may match both 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 unified with only one of them. For each choice, one ends up with a distinct database instance. Since a large database has often many potentially matches, the number of clean database instances may be enormous. It is *not* clear which clean instance one should use to learn an accurate definition for a target relation.

Recently, there have been efforts to develop systems that use data constraints to clean and materialize a unified database instance [9, 13, 59]. For instance, the HoloClean system uses data constraints, such as denial constraints and matching dependencies, to perform data repairing and materialize a probabilistic database [59]. To find repairs, HoloClean generates all possible values for an entity and assigns probabilities to them. These values are taken from tuples in the input database. To generate a unified database instance, for each entity with multiple possible values, HoloClean assigns the value with the highest probability. There are some downsides to this approach. First, HoloClean may produce inconsistent databases. For instance, in one tuple, the value *Star Wars* may be repaired with the value *Star Wars: Episode IV - 1977*, while in another tuple, the same value *Star Wars* may be repaired with the value *Star Wars: Episode III - 2005*. The generated database is inconsistent because the same entity in the original database is given two different interpretations. Another problem is that the data repairs considered by HoloClean are only given by the existing values in the database. In some cases, the correct data repair may not be an existing value in the database, but a combination of values or a repair given by a domain expert.

The aforementioned tasks substantially increase the time needed to get insights. In fact, most data scientists spend about 80% of their time on data preparation tasks, such as dealing with data heterogeneities [42]. Since users cannot wait for a long time or allocate enormous resources to prepare large datasets, researchers have proposed on-demand approaches to data preparation [36, 37, 42, 65]. These methods may reduce the time and effort of data cleaning by preparing only a subset of the data that may be relevant to the task at hand. More importantly, the process of data analysis is inherently iterative [37]. Users may start by collecting the data items which they deem relevant to the target relation, clean and prepare them, and learn a definition over the data. If the learned definition is *not* sufficiently accurate, users may look for other data items and repeat this process. An on-demand approach enables users to evaluate the relevance of a dataset to the target relation faster.

In this chapter, we build upon the idea of *on-demand* data analysis and propose a

novel learning method that learns directly over the original data without finding and materializing appropriate clean instance(s) of the data. Our approach advances the on-demand approaches as users do *not* have to perform any cleaning and preparation on the underlying data. Instead, users provide only a set of declarative (logical) constraints on the database schema, which determine values of what attributes can meaningfully match and be unified [5, 9, 11, 13, 30, 32, 39, 38, 64]. For instance, the values of the attribute *title* in relation *mov2totalGross* in BOM and the ones in the attribute *title* in relation *movies* in IMDb databases may match and be unified. On the other hand, it is *not* meaningful for the values of attributes *gross* in relation *mov2totalGross* and *year* in relation *movies* to be unified as they refer to different types of entities and values. Hence, our method substantially reduces the effort needed to produce clean database instance(s) for data analysis. The contributions of this chapter are the following:

1. Since the representational conflicts in a heterogeneous database are not resolved, the properties of a desirable learned definition over a heterogeneous database are *not* clear. In particular, one heterogeneous database may have several clean instances. We introduce and formalize the problem of learning over a content-heterogeneous relational database (Section 5.3).

2. We propose a novel relational learning algorithm called *CastorX*, an extension of Castor (Chapter 3), to learn over a database with heterogeneous values (Section 5.4). It leverages the set of declarative constraints on attributes whose values can meaningfully match and unify during learning to learn an effective definition.

3. Every learning algorithm chooses the final result based on its coverage of the training data, i.e., roughly speaking, the more (less) positive (negative) examples a definition covers the more accurate it is. It is challenging to perform these steps effectively over a database without cleaning it, as resolving certain entities, e.g., movies, may lead to unifying other entities of the same or other types, e.g., directors. Furthermore, one database may have numerous possible clean versions. We propose an efficient method to compute the coverage of a definition directly over the heterogeneous database without cleaning it (Section 5.4.2).

4. We provide an implementation of CastorX over a main-memory relational database management system. We use sampling techniques to scale learning over large

databases while keeping it effective (Section 5.5).

5. We perform an extensive empirical study over real-world datasets and show that CastorX scales to large databases and learns efficiently over heterogeneous databases.

5.2 Matching & Cleaning

5.2.1 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. If one wants to match and resolve values in a couple of attributes, one may use a supervised or unsupervised matching function to identify and unify their potential matches according to the domain of those attributes [24, 33]. For example, given that the domain of attributes is a set of strings, one may use string similarity functions, such as edit distance, to find potential matches. Users develop and use matching and resolution rules and functions based on their domain knowledge. For example, consider relation *Employee(id, name, home-phone, address)*. According to her domain knowledge, the user knows 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 large number of matching rules over several sets of attributes in a large relational database [5, 9, 11, 13, 30, 32, 39, 38, 64]. Furthermore, these rules may interact with each other. For example, consider again the relation *Employee(id, name, home-phone, address)*. Assume that the user also knows 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 apply the second rule to unify the values of *id* in these tuples. The abundance of rules to match values and their

interactions and dependencies call for a more formal approach to data cleaning where one can guarantee that all matching and resolution rules have been applied to the original data.

The database community has proposed declarative matching and resolution rules to express the domain knowledge about matching and resolution [5, 9, 11, 13, 30, 32, 39, 38, 64]. *Matching dependencies (MD)* are a popular type of such declarative rules, which provide a simple yet powerful method of expressing domain knowledge on matching values [10, 12, 30]. Let $\mathcal{S}$ be the schema of the original database and R_1 and R_2 two relations in $\mathcal{S}$. Let $dom(A)$ be the domain of values for attribute A. Attributes A_1 and A_2 from relations R_1 and R_2, respectively, are comparable if $dom(A_1) = dom(A_2)$. Given two pairs of pairwise comparable attributes A_1, A_2 and B_1, B_2 from relations R_1 and R_2, respectively, an MD ϕ in $\mathcal{S}$ is a sentence of the form $R_1[A_1] \approx R_2[A_2] \rightarrow R_1[B_1] \rightleftharpoons R_2[B_2]$, where $\approx$ is a similarity operator and $R_1[B_1] \rightleftharpoons R_2[B_2]$ indicates that the values of $R_1[B_1]$ and $R_2[B_2]$ refer to the same value, i.e., are interchangeable. Intuitively, the aforementioned MD says that given the values of $R_1[A_1]$ and $R_2[A_2]$ are sufficiently similar, the values of $R_1[B_1]$ and $R_2[B_2]$ are essentially different representations of the same value. For example, consider again the database that contains relations from *IMDb* and *BOM* whose schema fragments are shown in Table 5.1. According to our discussion in Section 5.1, one can define the following MD:

$$\textit{movies[title]} \approx \textit{highBudgetMovies[title]} \rightarrow \textit{movies[title]} \rightleftharpoons \textit{highBudgetMovies[title]}.$$

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. The definition of MDs extend for the case where A_1, A_2 are lists of attributes. For the sake of simplicity, we assume that MDs have only a single attribute in their left-hand sides. Our results extend to MDs with a list of attributes on their left-hand side. We also assume that all attributes share the same domain *dom* in the rest of this chapter. Our results generalize to other cases.

5.2.2 Stable Instances

Given an input database with MDs on its relations, one must enforce the MDs to generate a high-quality database. Let t_1 and t_2 belong to R_1 and R_2 in database I of schema $\mathcal{S}$, respectively, such that $t_1^I[A_1] \approx t_2^I[A_2]$. To enforce the MD $\phi : R_1[A_1] \approx R_2[A_2] \rightarrow R_1[B_1] \rightleftharpoons R_2[B_2]$ on I, one must make the values of $t_1^I[B_1]$ and $t_2^I[B_2]$ identical as they actually refer to the same value [12, 30]. For example, if attributes B_1 and B_2 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 $t_1^I[B_1]$ and $t_2^I[B_2]$ refer. The following definition formalizes the concept of applying an MD to the tuples t_1 and t_2 on I.

Definition 5.2.1. *Database I' of schema $\mathcal{S}$ is the immediate result of enforcing MD ϕ on t_1 and t_2 in I, denoted by $(I, I')_{[t_1,t_2]} \models \phi$ if*

1. *$t_1^I[A_1] \approx t_2^I[A_2]$, but $t_1^I[B_1] \neq t_2^I[B_2]$;*
2. *$t_1^{I'}[B_1] = t_2^{I'}[B_2]$; and*
3. *I and I' agree on every other tuple and attribute value.*

One may define a matching function over some domains to map the values that refer to the same value or entity to the correct value or entity in the stable instance. It may, however, be difficult to define such a function in many cases due to the lack of information. For example, let B_1 and B_2 in Definition 5.2.1 contain information about names of people and $t_1^I[B_1]$ and $t_2^I[B_2]$ have values *J. Bryan Smith* and *John B. Smith*, respectively, which according to an MD refer to the same actual name. It is *not* clear how to compute this name using the values of $t_1^I[B_1]$ and $t_2^I[B_2]$. Thus, we know that after enforcing ϕ, the values of $t_1^{I'}[B_1]$ and $t_2^{I'}[B_2]$ will be identical, but we do *not* know their exact values. Because we aim at developing learning algorithms that are efficient and effective over databases from various domains, we do *not* fix any matching method in this chapter. We assume that matching every pair of values a and b in the database creates a fresh and new value denoted as $v_{a,b}$.

Given the database I of schema $\mathcal{S}$ with set of MDs Σ, I' is *stable* if $(I, I')_{[t_1,t_2]} \models \phi$ for all $\phi \in \Sigma$ 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 the heterogeneities that may reduce the effectiveness of learning and analyzing databases. 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 5.2.1 to the output of the previous application finitely many times. In other words, let $I, I_1, \ldots, I_k$ denote the sequence of databases produced by applying MDs according to Definition 5.2.1 starting from I such that I_k is stable. We say that (I, I_k) satisfy Σ and denote it as $(I, I_k) \models \Sigma$. Each database may have (finitely) many stable instances depending on the order of applications of MDs [12, 30].

Example 5.2.2. *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 5.1. Consider MD* $movies[title] \approx highBudgetMovies[title] \rightarrow movies[title] \rightleftharpoons highBudgetMovies[title]$. *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 unification alternative leads to a distinct stable instance.*

The number of stable instances of a database generally grows exponentially with the number of matches and unification applications in the database. For example, in Example 5.2.2, if relation *highBudgetMovies* has a hundred titles that each matches and unifies with at least two titles in relation *movies*, the database will have around 2^{100} stable instances. Since there are often many matches and unification applications in a large database, one may have to generate numerous stable and clean databases. To produce only one stable instance, one may assume that all distinct unified values for a given value in the database represent the same value. For example, in Example 5.2.2, if one assumes that all movies with prefix *Star Wars* represent the same movie, one can unify the titles of all movies to the same value. Nevertheless, this approach often violates the semantics of the underlying database and adds misleading information to it. For example, in Example 5.2.2, this approach leads to assuming that movies with titles 'Star Wars: Episode IV - 1977' and 'Star Wars: Episode III - 2005' have the same titles, which is obviously incorrect and misleading. This approach may also violate the integrity constraints in the database. For example, if matches and unifications are between key

attributes, this approach will violate the key or functional dependency constraint by adding duplicate keys to a relation and make the database inconsistent.

5.3 Learning Over Heterogeneous Data

5.3.1 Approaches to Learning Over Heterogeneous Data

Given the database I of schema $\mathcal{S}$ with MDs Σ and a set of training examples E, we wish to learn a Horn definition for a target relation T in terms of the relations in schema $\mathcal{S}$. Obviously, one may *not* learn an accurate definition by applying current learning algorithms over the set of input databases as it may consider different occurrences of the same entity or value to be distinct because of their different representations. Let *StableInstances*(I, Σ) be the set of stable instances of I. One can learn definitions by generating all stable instances, i.e., $\mathbf{J} = \mathit{StableInstances}(I, \Sigma)$, learning a definition over each stable instance $J \in \mathbf{J}$ separately, and computing a union (disjunction) of all learned definitions. Since the discrepancies in value representations are resolved in the stable instances, this approach may learn accurate definitions.

However, this method is *not* desirable or feasible for learning over large databases. As a large database may have numerous stable instances, 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 stable instance, which may take an extremely long time. More importantly, as the learning has been done separately over each stable instance, 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 stable instances 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 a lot of 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 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 the most positive and the least negative examples over I_2^s and I_1^s. Finally, it is *not* clear how to encode and represent usably the final result as we may end up with numerous different definitions, each of which is accurate over one stable instance. One may end up with hundreds or thousands of learned definitions. Thus,

learning over each stable instance may take a very long time, *not* return an effective definition that considers information of all stable instances, and will be hard to use.

Another approach is to consider only the information shared among all stable instances for learning. The resulting definition will cover the most positive and the least negative examples considering the information common among all clean instances. This idea has been used in the context of query answering over inconsistent data [6, 12]. However, this approach may lead to ignoring many positive and negative examples 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 5.2.2. 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. Nevertheless, the training examples are very important in delivering an effective result and are usually costly to attain. In a sufficiently heterogeneous database, most or all positive and negative examples will *not* have the same information among all stable instances. The learning algorithm may simply learn an empty definition in these cases.

Thus, we hit a middle-ground. We follow the approach of learning directly over the original database. But, 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 stable instances. If one increases the expressivity of the language, learning and checking coverage for each clause may become inefficient, e.g., disjunctive Datalog [25]. We ensure that the added capability to the language of definitions is minimal so learning remains efficient. In this section, we present our modifications to the hypothesis space and semantic of learning.

5.3.2 Representing Heterogeneity in Definitions

We represent the heterogeneity of the underlying data in the language of the learned definitions. Each new definition encapsulates the varieties of definitions learned over the stable instances of the underlying database. To achieve this, we increase the hypothesis space of relational learning algorithms over schema $\mathcal{S}$ by introducing a fresh and unique (built-in) relation symbol per each MD in $\mathcal{S}$. These symbols do *not* belong to $\mathcal{S}$ and are

called *matching relations*. More precisely, given schema $\mathcal{S}$ and MD ϕ in $\mathcal{S}$, we add relation symbol m_ϕ with arity two to the set of relation symbols used by the Datalog definitions in the hypothesis space. A literal with a matching relation symbol is a *matching literal*. Consider again the database created by integrating IMDb and BOM datasets, whose schema fragments are in Table 5.1, with MD ϕ : *movies[title]* $\approx$ *highBudgetMovies[title]* $\rightarrow$ *movies[title]* $\rightleftharpoons$ *highBudgetMovies[title]*. We may learn the following Datalog definition for the target relation *highGrossing*.

$$highGrossing(x) \leftarrow movies(y, t, z), mov2genres(y, \text{`}comedy\text{'}),$$
$$highBudgetMovies(x), m_\phi(x, t).$$

The matching literal $m_\phi(x, t)$ represents the relationship between titles in relations *highGrossing* and *movies*. According to the definition of MDs, for every matching literal m_ϕ, we have $m_\phi(x, y)$ if and only if $m_\phi(y, x)$.

We call a clause (definition) *stable* if it does *not* have any matching literal. Each clause (definition) with matching literals represents a set of stable clauses (definitions). We convert a clause with matching literals to a set of stable clauses by iteratively applying the MDs that correspond to each matching literal and eliminating the matching literal from the clause, similar to the process of applying MDs to a database instance described in Section 5.2. Given a clause C and an MD ϕ, to apply ϕ to C, we do the following. For each matching literal $m_\phi(x, y)$ in C, we create a new variable $v_{x,y}$. We then replace every occurrence of x and y in C with $v_{x,y}$. Finally, we remove the original matching literal $m_\phi(x, y)$ from C. The algorithm progressively applies MDs to the created clause until no matching literal is left in the clause. Similar to the one explained in Section 5.2, this algorithm is guaranteed to terminate. The resulting set is called the *stable clauses* of the input clause.

Example 5.3.1. *Consider the following clause over the movie database of IMDb and BOM.*

$$highGrossing(x) \leftarrow movies(y, t, z), mov2genres(y, \text{`}comedy\text{'}),$$
$$highBudgetMovies(x), m_\phi(x, t).$$

The application of MD ϕ : $highGrossing[title] \approx movies[title] \rightarrow highGrossing[title] \rightleftharpoons$

movies[title], unifies variables t and x to variable $v_{x,t}$ and generates the following clause:

$$highGrossing(v_{x,t}) \leftarrow movies(y, v_{x,t}, z), mov2genres(y, \text{'}comedy\text{'}), highBudgetMovies(v_{x,t}).$$

After each application of an MD to a clause, the modified variables will *not* be similar to any term in the clause because we do *not* have enough information to declare the variable or constant created after matching equal to any other variable or constant. Therefore, we treat the application of MDs to clauses similarly to how we treat the application of MDs to databases explained in Section 5.2. Similar to the application of MDs to a database instance, the application of MDs to a clause may generate multiple stable clauses.

Example 5.3.2. *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 $H : T(x) \leftarrow R(y), m_{\phi_1}(x, y), S(z), m_{\phi_2}(x, z)$ over this schema has two stable definitions: $H'_1 : T(v_{x,y}) \leftarrow R(v_{x,y}), S(z)$ and $H'_2 : T(v_{x,z}) \leftarrow R(y), S(v_{x,z})$.*

As Example 5.3.2 illustrates, using matching literals enables us to provide a compact representation of multiple learned clauses and definitions where each may explain the patterns in the training data in some stable instances of the input database. Given an input definition H, the *stable definitions* of H are a set of definitions where each definition contains exactly one stable clause per each clause in H.

5.3.3 Coverage Over Heterogeneous Data

A learning algorithm evaluates the score of a definition according to the number of its covered positive and negative examples. One way to measure the score of a definition is to compute the difference of the number of positive and negative examples covered by the definition. Each definition may have multiple stable definitions each of which may cover a different number of positive and negative examples on stable instances of the underlying database. Thus, it is *not* clear how to compute the score of a definition over a database.

One approach is to consider that a definition covers a positive example if at least one of its stable definitions covers it in some stable instance(s). Given all other conditions

are the same, this approach may lead to learning a definition with numerous stable definitions where each may *not* cover sufficiently many or none positive examples. Thus, it is *not* clear whether each stable definition is accurate. A more restrictive approach is to consider that a definition covers a positive example if all its stable definitions cover the example. This method will deliver a definition whose stable definitions have high positive coverage over stable instances. There are similar alternatives for defining coverage of negative examples. One may consider that a definition covers a negative example if all its stable definitions cover it. Thus, if at least one stable definition does *not* cover the negative example, the definition will *not* cover it. This approach may lead to learning numerous stable definitions, which cover many negative examples. On the other hand, a restrictive approach may define a negative example covered by a definition if at least one of its stable definitions covers it. In this case, generally speaking, each learned stable definition will *not* cover too many negative examples.

We follow the more restrictive approach for definition of coverage of positive and negative examples.

Definition 5.3.3. *A definition H covers a positive example e with regard to database I if and only if every stable definition of H covers e in some stable instance of I.*

Example 5.3.4. *Consider again the schema, MDs, and definition H in Examples 5.3.2 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 5.3.3 as its stable definitions H'_1 and H'_2 cover the training example in stable instances I'_1 and I'_2, respectively.*

Definition 5.3.3 provides a more flexible semantic than considering only the common information between all stable instances as described in Section 5.3.1. The latter semantic considers that the definition H covers a positive example if it covers the example in all stable instances of a database. As explained in Section 5.3.1, this approach may lead to ignoring many if not all examples for learning.

Definition 5.3.5. *A definition H covers a negative example e with regard to database I if at least one of the stable definitions of H covers e in some stable instance of I.*

To compute the coverage of a clause C for a set of positive examples, learning algorithms simply check the coverage of the clause for each example in the set and sum the covered examples. This approach can be safely applied for the case where C is stable. However, if C is *not* stable, this method may deliver a clause that does *not* cover sufficiently many positive examples in any stable instance. For example, consider a database with one hundred stable instances and a set of one hundred positive examples. Let each stable clause of a clause C cover only one example over each stable instance of the database. Obviously, C is *not* effective on any of these instances as its stable clauses cover an extremely small number of positive examples over each stable instance. This method, however, declares that C covers all the positive examples and is effective. Thus, a more reasonable approach is to ensure that each stable clause of C also covers sufficiently many positive examples over a stable instance. Given database I, stable clause C, and example e, let $\mathcal{I}_{C,e}$ be the set of stable instances of I on which C covers e.

Definition 5.3.6. *Given a database I, clause C, and set of positive examples E^+, C covers E^+ in some stable instance if C covers each $e \in E^+$ and for every stable clause C_s of C and every pair of examples $e, f \in E^+$, we have $\mathcal{I}_{C_s,e} \cap \mathcal{I}_{C_s,f} \neq \emptyset$.*

The definition naturally extends to (Horn) definitions and the set coverage for negative examples.

As explained in Section 5.3.1, our semantic of coverage based on Definitions 5.3.3 and 5.3.5 provides a middle-ground between the approaches of using only the information common between all stable instances and learning over each stable instance separately and returning the union of the results. It avoids ignoring too many examples and also ensures that the learned definition does *not* cover too many negative examples overall. It also guarantees that each stable definition of the learned definition covers sufficiently many positive examples and does *not* cover too many negative examples if the learning algorithm checks these restrictions on the learned definition.

5.4 CastorX: A Learning Algorithm for Heterogeneous Data

There are, generally speaking, two types of relational learning algorithms. In the top-down approach [57, 60, 67], algorithms search the hypothesis space from general to specific. A top-down algorithm starts with an empty clause, iteratively adds some lit-

erals to it, evaluates the resulting clause(s) over the training examples and database, and stops as soon as the clause satisfies certain criteria, e.g., covers the most positive and least negative examples. This method is challenging to use for learning over heterogeneous data, as it has to evaluate numerous clauses, i.e., queries, over the original (dirty) dataset. Since the dataset may have many stable solutions, query answering is shown to be generally difficult and inefficient over a relatively large and heterogeneous dataset [12]. The second approach for relational learning algorithms is bottom-up, in which the algorithm starts by exploring the patterns and clauses available in the data and then generalizes them to cover the training examples [46, 50, 54]. More specifically, a bottom-up algorithm 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 most positive and as least negative examples as possible.

In this section, we propose a bottom-up algorithm called CastorX for learning over heterogeneous data efficiently. CastorX is an extension of Castor (Chapter 3). To learn over heterogeneous data, CastorX integrates the input MDs into the learning process. It also uses novel techniques to pick the most accurate clauses (and definitions) without computing the stable instances of the data. Because it is guided by the training data, CastorX automatically determines whether and which databases one has to integrate to learn an accurate definition for the target concept. Next, we explain the two main steps of the algorithm in detail.

5.4.1 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. The concept of a bottom-clause was introduced in Section 3.5. In this chapter, extend the bottom-clause construction algorithm to leverage matching dependencies. Let I be the input database of schema $\mathcal{S}$ and Σ be the input matching dependencies. The bottom-clause construction algorithm consists of two phases. First, CastorX 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 (similarity) search operations, starting from e. Given the information relevant to e, CastorX creates the bottom-clause C_e.

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)	spanishMovies(m3)

Table 5.2: Example movie database.

Example 5.4.1. *Given example highGrossing(Superbad), the database in Table 5.2, and MD*

$$\phi : highGrossing[title] \approx movies[title] \rightarrow highGrossing[title] \rightleftharpoons movies[title],$$

CastorX 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, e.g., Superbad, does not match with the movie title in the movies relation, e.g., Superbad (2007), the tuple movies(m1, Superbad (2007), 2007) is obtained through a similarity search according to MD ϕ. The other tuples are obtained through exact search.

To find the information relevant to e, CastorX uses Algorithm 10. CastorX maintains a set M that contains all seen constants. Let $e = T(a_1, \dots, a_n)$ be a training example. First, CastorX adds $a_1, \dots, a_n$ to M. These constants are values that appear in tuples in I. Then, CastorX searches all tuples in I that contain at least one constant in M and adds them to I_e. For exact search, CastorX uses simple *select* operations. For similarity search, CastorX uses MDs in Σ. If M contains constants in some relation R_i and given an MD $\phi \in \Sigma$, $\phi : R_i[A] \approx R_j[B] \rightarrow R_i[A] \rightleftharpoons R_j[B]$, CastorX performs a similarity search over $R_j[B]$ to find relevant tuples in R_j, denoted by $\psi_{B \approx M}(R_j)$. For each new tuple in I_e, the algorithm extracts new constants and adds them to M. The algorithm repeats this process for a fixed number of iterations d.

To create the bottom-clause C_e from I_e, CastorX 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 $t \in I_e$, CastorX

Algorithm 10: CastorX's bottom-clause construction algorithm.

```
Input  : example e, # of iterations d, sample size s
Output: bottom-clause C_e
I_e = {}
M = {} // M stores known constants
add constants in e to M
for i = 1 to d do
    foreach relation R ∈ I do
        foreach attribute A in R do
            I_R = σ_{A∈M}(R)
            if ∃ MD φ ∈ Σ, φ : S[B] ≈ R[A] → S[B] ⇌ R[A] then
                I_R = I_R ∪ ψ_{A≈M}(R)
            foreach tuple t ∈ I_R do
                add t to I_e and constants in t to M
C_e = create clause from e and I_e
return C_e
```

creates a literal and adds it to the body of the clause, replacing each constant in t with its assigned variable. If there is an input MD $\phi : R_i[A] \approx R_j[B] \rightarrow R_i[A] \rightleftharpoons R_j[B]$, and there is a tuple $t' \in R_i$, then I_e may contain a tuple t obtained through similarity search. In this case, we add a *matching literal* $m_\phi(v_1, v_2)$, where v_1 and v_2 are the variables assigned to $t'[A]$ and $t[B]$, respectively. A matching literal indicates that the two values represented by the variables in the literal are interchangeable, according to the MD.

Example 5.4.2. *Given the relevant tuples found in Example 5.4.1, CastorX creates the following bottom-clause:*

$$\begin{aligned} \mathit{highGrossing}(x) \leftarrow & \mathit{movies}(y,t,z), m_\phi(x,t), \mathit{mov2genres}(y, \text{`}\mathit{comedy}\text{'}), \\ & \mathit{mov2countries}(y,v), \mathit{countries}(v, \text{`}\mathit{USA}\text{'}), \mathit{englishMovies}(y), \\ & \mathit{mov2releasedate}(y, \text{`}\mathit{August}\text{'}, u). \end{aligned}$$

5.4.2 Generalization

After creating the bottom-clause C_e for example e, CastorX generalizes C_e iteratively. To generalize C_e, CastorX randomly picks a subset $E^{+s} \subseteq E^+$ of positive examples. For

each example e' in E^{+s}, CastorX generalizes C_e to generate a candidate clause C', which is more general than C_e and covers e'. It does so by removing the *blocking literals*. Let $C_e = T \leftarrow L_1, \cdots, L_n$ be the bottom-clause. L_i is a *blocking literal* if and only if i is the least value such that for all substitutions θ where $e' = T\theta$, the clause $(T \leftarrow L_1, \cdots, L_i)\theta$ does not cover e' [50]. From the set of generalized clauses, CastorX selects the highest scoring candidate clause. The score of a clause is the number of positive examples covered by the clause minus the number of negative examples covered by the clause. It then repeats this process with the selected clause until the clause cannot be improved.

Example 5.4.3. *Consider the bottom-clause C_e in Example 5.4.2 and positive example e' = highGrossing('Zoolander'). To generalize C_e to cover e', CastorX drops the literal mov2releasedates$(y, \text{'August'}, u)$ because the movie Zoolander was not released in August.*

A matching literal can also be a blocking literal.

Example 5.4.4. *Consider again the bottom-clause C_e in Example 5.4.2 and positive example e'' = highGrossing('Inception'). The movie with title 'Inception' does not appear in the movies relation. Therefore, the first blocking literal in C_e is $m_\phi(x, t)$, which is in the join path between highGrossing(x) and movies(y, t, z).*

CastorX checks whether a candidate clause covers training examples in order to find 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 [22]. One approach to perform a coverage test is to transform the clause into a 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., hundreds of literals, 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 [12]. Moreover, it is *not* clear how to evaluate clauses that contain matching literals.

Therefore, CastorX uses an approach based on θ-subsumption. The concept of θ-subsumption was introduced in Section 2.1. We review it in this section and extend to handle non-stable clauses. Assume that clause C does not contain matching literals, i.e., C is a stable clause. Clause C *θ-subsumes* clause G, denoted by $C \subseteq_\theta G$, if and only if there is some substitution θ such that $C\theta \subseteq G$ [2, 22]. $C\theta \subseteq G$ means that the result of

applying substitution θ to clause C is a subset of clause G. To evaluate whether a clause C covers an example e over database I, we first build a ground bottom-clause G_e for e in I, i.e., a bottom-clause for e without replacing constants with variables. Then, we check whether $C \subseteq_\theta G_e$, which in turn indicates whether $C \wedge I \models e$. This approach is used by other learning algorithms, such as Castor [50, 54]. Ideally, G_e must have one literal per each tuple in the database that is connected to e through some joins; otherwise, the θ-subsumption test may declare that C does *not* cover e when C actually covers e. Ground bottom-clauses may be very large over large databases. The θ-subsumption is expensive over large bottom-clauses. We will discuss how to handle these cases in Section 5.5.

Clause C and ground bottom-clause G_e may *not* be stable. Therefore, there are possibly multiple stable clauses that can be derived from C and G_e. Following Definitions 5.3.3 and 5.3.5, we define θ-subsumption between potentially non-stable clauses.

Definition 5.4.5. *Given clauses C and G_e^+, where G_e^+ is the ground bottom-clause for positive example e, clause C θ-subsumes G_e^+ if each stable clause of C θ-subsumes at least one stable clause of G_e^+.*

Definition 5.4.6. *Given clauses C and G_e^-, where G_e^- is the ground bottom-clause for negative example e, clause C θ-subsumes G_e^- if at least one stable clause of C θ-subsumes at least one stable clause of G_e^-.*

According to Definitions 5.4.5 and 5.4.6, given training example e, to check whether C θ-subsumes G_e^+ (G_e^-), one has to enumerate and check θ-subsumption of almost every pair of stable clauses of C and G_e^+ (G_e^-) in the worst case. Since both (ground) bottom-clauses normally contain many literals and θ-subsumption is NP-hard [2], this method is time-consuming. More importantly, because the learning algorithm performs numerous coverage tests, learning a definition may be extremely inefficient.

Actually, we can check whether C θ-subsumes G_e^+ (G_e^-) in a much more efficient way and without enumerating all stable clauses of C and G_e^+ (G_e^-). We consider matching literals in C and G_e^+ (G_e^-) that correspond to the same MD as the same relation symbol. For positive coverage, to find a substitution θ for C such that $C\theta \subseteq G_e^+$, we treat matching literals in C and G_e^+ as normal literals. If such substitution exists, then C θ-subsumes G_e^+, hence C covers e.

Example 5.4.7. *Consider clause C*

$$highGrossing(x) \leftarrow movies(y,t,z), m_\phi(x,t), mov2genres(y, \text{`}comedy\text{'}).$$

Consider positive example $e' = highGrossing(Zoolander)$ *and ground bottom-clause* $G^+_{e'}$

$$\begin{aligned} highGrossing(\text{`}Zoolander\text{'}) \leftarrow\ & movies(\text{`}m2\text{'}, \text{`}Zoolander(2001)\text{'}, \text{`}2001\text{'}),\\ & m_\phi(\text{`}Zoolander\text{'}, \text{`}Zoolander(2001)\text{'}), mov2genres(\text{`}m2\text{'}, \text{`}comedy\text{'}),\\ & mov2countries(\text{`}m2\text{'}, \text{`}c1\text{'}), countries(\text{`}c1\text{'}, \text{`}USA\text{'}),\\ & englishMovies(\text{`}m2\text{'}), mov2releasedate(\text{`}m2\text{'}, \text{`}September, 2001\text{'}).\end{aligned}$$

C θ-subsumes $G^+_{e'}$ using substitution $\theta = \{x = \text{`}Zoolander\text{'}, t = \text{`}Zoolander(2001)\text{'}, y = \text{`}m2\text{'}, z = \text{`}2001\text{'}\}$. Thus, C covers e'.

We follow a similar approach for the negative examples. Let G^-_e be the ground bottom-clause for the negative example e. We generate all stable clauses of C as described in Section 5.3 and check whether the generated stable clause θ-subsumes G^-_e exactly the same way as checking θ-subsumption for C and a ground bottom-clause for a positive example explained in the preceding paragraph. We declare the C θ-subsumes G^-_e as soon as one stable clause of C θ-subsumes G^-_e. In the following theorem, we prove that the aforementioned algorithms correctly detects θ-subsumption between clauses.

Theorem 5.4.8. *Given a clause C and G^+_e over schema $\mathcal{S}$, clause C θ-subsumes G^+_e if and only if the aforementioned algorithm for checking the θ-subsumption of C and a positive ground bottom-clause finds a substitution θ such that $C\theta \subseteq G^+_e$. Similarly, given a clause C and G^-_e over $\mathcal{S}$, clause C θ-subsumes G^-_e if and only if the aforementioned algorithm for checking the θ-subsumption of C and a negative ground bottom-clause finds a substitution θ for one stable clause of C, C_s such that $C_s\theta \subseteq G^-_e$.*

Proof. We prove the theorem for ground bottom clauses of positive examples. Assume that there is a substitution θ such that $C\theta \subseteq G^+_e$. Thus, each matching literal of C, such as $m_\phi(x,y)$, is mapped and unified with a matching literal $m_\phi(x',y')$ in G^+_e. Let C_ϕ and $G^+_{e\phi}$ be the results of applying $m_\phi(x,y)$ and $m_\phi(x',y')$ to C and G^+_e, respectively. We show that the applications of mapped matching literals in these clauses preserve the substitution, i.e., $C_\phi\theta \subseteq G^+_{e\phi}$. Applying a matching literal does *not* eliminate any

non-matching literal from a clause and only replaces some of their variables with new ones. Let C_ϕ and $G^+_{e\phi}$ be created by applying $m_\phi(x, y)$ and $m_\phi(x', y')$ to $R(\bar{x}), S(\bar{y})$ and $R(\bar{x'}), S(\bar{y'})$, respectively. The application of $m_\phi(x, y)$ replaces x and y in $R(\bar{x})$ and $S(\bar{y})$ with a new variable $v_{x,y}$. Similarly, by applying $m_\phi(x', y')$, we replace x' and y' in $R(\bar{x'})$ and $S(\bar{y'})$ with $v_{x',y'}$. The applications of $m_\phi(x, y)$ and $m_\phi(x', y')$ do *not* modify the literals that do *not* share any variable with them. Thus, we have $C_\phi\theta \subseteq G^+_{e\phi}$. Furthermore, as we assume that matching functions are *not* similarity preserving, i.e., each application of a matching function creates a fresh constant, subsequent applications of the matching literals other than $m_\phi(x, y)$ and $m_\phi(x', y')$ do *not* modify the variables introduced by applying these matching literals. Thus, the substitution θ is preserved over the application of each matching literal in C and its corresponding one in G^+_e. After exhausting the application of all matching literals in C and their corresponding ones in G^+_e, G^+_e may still contain matching literals. However, applying these literals to G^+_e retains the substitution between the stable clause of C and subsequent results of application of produced stable clauses of G^+_e. The necessity of the theorem is proved by reversing the aforementioned steps. Now, we consider the case that G^-_e is a negative ground bottom-clause. In this case, the algorithm checks the subsumption using every stable clause of the bottom-clause. Let C be one of these stable clauses. If C subsumes G^-_e, according to the first part of this proof, C subsumes every stable clause of G^-_e. The proof for the necessity of this part is also similar to the one of the part of the theorem on positive ground bottom-clauses. □

Given bottom-clause C and a set of positive examples E^+, according to Definition 5.3.6, it seems that one *cannot* use the simple method of computing the coverage of C for each example in E^+ and summing up the results to calculate the coverage of C over E^+. However, in the following theorem we prove that if one uses the aforementioned subsumption method to check the coverage of C for each example, this simple method returns the desired coverage value for C. Therefore, our algorithm is able to compute the coverage of a clause over a set of examples without any extra overhead. In the following theorem, we assume that no MD is defined over the relation that contains the training examples, i.e., all MDs are defined on (other) relations in the background knowledge.

Theorem 5.4.9. *Given a database I, set of positive examples E^+, and clause C, let C θ-subsume G^+_e and G^+_f, which are ground bottom-clauses for $e, f \in E^+$, respectively. For*

each stable clause C_s of C, we have $\mathcal{I}_{C_s,e} \cap \mathcal{I}_{C_s,f} \neq \emptyset$.

Proof. For brevity, we denote G_e^+ and G_e^+ as G_e and G_f, respectively. Our goal is to show that the set $\mathcal{I}_{C_s}$ of stable instances in which C_s θ-subsumes some stable clauses of both G_e and G_f is not empty. Let G_e^s and G_f^s be the stable clauses of G_e and G_f, respectively, that are θ-subsumed by C_s. The set $\mathcal{I}_{C_s}$ becomes empty if and only if a tuple in the original database has different values in the stable instance(s) in which C θ-subsumes G_e^s compared to all stable instance(s) in which C θ-subsumes G_f^s. If this holds, there will not be any stable instance in which C_s θ-subsumes both G_e^s and G_f^s. Otherwise, one can construct at least one stable instance over which C_s θ-subsumes both G_e^s and G_f^s.

First, assume that G_e and G_f do *not* share any normal, i.e., non-matching, literal. Since G_e and G_f do *not* have any literal in common, G_e^s and G_f^s are produced by applying matching literals on two completely different sets of literals. Let M_1 and M_2 be the sequence of MDs that correspond to the matching literals whose applications on G_e and G_f create the stable clauses G_e^s and G_f^s, respectively. Since C_s θ-subsumes both G_e^s and G_f^s, $\mathcal{I}_{C_s,e}$ and $\mathcal{I}_{C_s,f}$ contain stable instances produced by applying the MDs that correspond to M_1 and M_2, respectively. There is *not* any tuple in the original database in which an MD from the MDs in M_1 and an MD from the MDs in M_2 are applied. Hence, the intersection of $\mathcal{I}_{C_s,e}$ and $\mathcal{I}_{C_s,f}$ is not empty and contain all the stable instances in which C_s θ-subsumes both G_e^s and G_f^s.

Now, assume that G_e and G_f share a literal $R(\overline{a})$. The sets $\mathcal{I}_{C_s,e}$ and $\mathcal{I}_{C_s,f}$ are disjoint if and only if 1) G_e and G_f have matching literals $m_1(b,c)$ and $m_2(b,d)$, $c \neq d$, respectively, that are used to create both G_e^s and G_f^s and 2) $m_1(b,c)$ is applied to literals $(R(\overline{a}), S(\overline{f}))$ in G_e and $m_2(b,d)$ is applied to literals $(R(\overline{d}), U(\overline{g}))$ in G_f. This way the tuple in the original database that corresponds to the literal $R(\overline{a})$ will have different constants in the stable instance(s) in which C θ-subsumes G_e^s than the ones in which C θ-subsumes G_f^s.

For the sake of simplicity, let $R(\overline{a})$ be the only literal shared between the bodies of G_e and G_f. Our proof extends to other cases. According to the algorithm of creating ground bottom-clauses, all literals in G_e that have a constant in common with $R(\overline{a})$, i.e., their corresponding tuples in the database are connected to the tuple of $R(\overline{a})$ in the data through some join paths, will also appear in G_f. The same claim is true about the

literals constructed and added to the ground bottom-clauses based on the tuples that are connected to $R(\overline{a})$ via the similarity predicate of some MD. Let us call this set of literals and their corresponding matching literals $L_{R(\overline{a})}$. According to the algorithm of ground bottom-clause construction, we may extend $L_{R(\overline{a})}$ to another set L, which contains all and only the literals that are connected to at least one literal in L via a join path, i.e., common constant, or a matching literal. We also add the matching literals that connect the (normal) literals of L to L. The literals in L appear in the bodies of both G_e and G_f.

Let θ_e and θ_f be the substitution mappings from constants in the literals of G_e and G_f, respectively, to the variables or constants in the literals of C. Members of θ_e are pairs of (a, x) where a is a constant in G_e and x is a variable or constant in C. Let $\theta_e^L \subseteq \theta_e$ and $\theta_f^L \subseteq \theta_f$ be the substitution mappings from G_e and G_f to C, respectively. Let's replace θ_f^L with θ_e^L in θ_f and get a substitution θ'_f. Since the literals in L do not have any constant or matching literal in common with the rest of literals in G_f, C still θ-subsumes G_f using θ'_f. θ_e and θ'_f unify every (matching) literal in L with the same (matching) literal in C.

Matching literals that are not in L do not share any constant with the literals in L. Hence, applying these matching literals does not change the constants of any literal in L. Thus, without loss of generality we assume that G_e and G_f contain only literals that appear in L. Let M_C be the sequence of matching literals in C whose application creates C_s. Let M_L be the sequence of matching literals in L that are unified with M_C using θ_e and θ'_f. One can generate G_e^s and G_f^s by first applying M_L and then the rest of matching literals in L in an arbitrary order. C_s θ-subsumes both G_e^s and G_f^s. Since G_e^s and G_f^s are created by applying the same sequence of matching literals on the same set of literals, $R(\overline{a})$ has the same value in both stable clauses. $\square$

Theorem 5.4.9 extends for negative examples. After learning, one may use these coverage testing methods to use the learned definition and apply the learned definition to do prediction over the original database.

5.4.3 Commutativity Property

As discussed in Section 5.3, our proposed semantic for coverage over heterogeneous data as described in Definitions 5.3.3 and 5.3.5 is more restrictive than the method of learning definitions over every stable instance and then providing a union of all definitions as the final result. Since the method of learning over every stable instance separately contains all possible definitions learned over the data, it is important to understand whether our algorithm works with the same set of (stable) definitions as this method to get an understanding of the coverage of the definitions produced by our algorithm in terms of the learned definitions over the individual stable instances. We show that our algorithm starts with and generalizes the same set of stable definitions used to learn over each stable instance. In other words, assume that we apply the bottom-clause construction part of our learning algorithm, as described in Section 5.4.1 directly to each stable instance of database I and output a set of stable bottom-clauses B. As we explained in the beginning of this section, the algorithm of CastorX over non-heterogeneous, i.e., stable, databases is essentially existing state-of-the-art relational learning algorithms [50, 54]. Now, assume that we apply the bottom-clause construction part of our proposed algorithm directly over the original heterogeneous database and get a (unstable) definition H. We show that B is exactly the set of stable instances of H. Let $\textit{StableDefinitions}(H)$ denote the set of all stable definitions of H. Let $\textit{BottomClause}(I, e, \Sigma)$ denote the bottom-clause generated by applying the bottom-clause construction algorithm using example e over (heterogeneous) database I with the set of MDs Σ. Also, let $\textit{BottomClause}_s(\textit{StableInstances}(I, \Sigma), e)$ be the set of stable clauses generated by applying the bottom-clause construction algorithm to every stable instance of I using example e.

Theorem 5.4.10. *Given database I with MDs Σ and set of positive examples E^+, for every positive example $e \in E^+$*

$$\textit{BottomClause}_s(\textit{StableInstances}(I, \Sigma), e) = \textit{StableDefinitions}(\textit{BottomClause}(I, e, \Sigma))$$

Proof. Without loss of generality, assume that all learned definitions contain one clause. Let $\mathbf{J} = \textit{StableInstances}(I, \Sigma) = \{J_1, \ldots, J_n\}$. We show that $\textit{BottomClause}(I, e, \Sigma) = C$ is a compact representation of $\textit{BottomClause}_s(\mathbf{J}, e) = \{C_1, \ldots, C_n\}$.

Let *StableDefinitions*$(C) = \{C'_1, \ldots, C'_m\}$. We remove the literals that are not head-connected in each clause in $\{C'_1, \ldots, C'_m\}$. Let $\{J^{C'_1}, \ldots, J^{C'_m}\}$ be the canonical database instances of $\{C'_1, \ldots, C'_m\}$ [2]. This set is the same set as the one generated by applying *StableInstances*(I^C, Σ), where I^C is the canonical database instance of C.

Let $\{J^{C_1}, \ldots, J^{C_n}\}$ be the canonical database instances of $\{C_1, \ldots, C_n\}$. By definition, I^C contains all tuples that are related to e, either by exact or similarity matching (according to MDs in Σ). Because *StableInstances*$(I^C, \Sigma) = \{J^{C'_1}, \ldots, J^{C'_m}\}$, all tuples that may appear in an instance in $\{J^{C_1}, \ldots, J^{C_n}\}$ must also appear in an instance in $\{J^{C'_1}, \ldots, J^{C'_m}\}$.

A tuple t may appear in an instance in $J^{C'_j} \in \{J^{C'_1}, \ldots, J^{C'_m}\}$, but not appear in the corresponding instance $J^{C_i} \in \{J^{C_1}, \ldots, J^{C_n}\}$. In this case, t became disconnected from training example e when generating the stable instance J_i, which is a superset of J^{C_i}. Then, when building bottom-clause C_i from J_i, a literal was not created for t. However, the same tuple would also become disconnected from training example e in $J^{C'_j}$. Because we remove literals that are not head-connected in each clause in $\{C'_1, \ldots, C'_m\}$, we would remove t from C'_j.

The sets of canonical database instances $\{J^{C'_1}, \ldots, J^{C'_m}\}$ and $\{J^{C_1}, \ldots, J^{C_n}\}$ are both generated using the function *StableInstances* with the same matching dependencies Σ, and only contain tuples related to e. Therefore, *StableDefinitions*$(C) = \{C'_1, \ldots, C'_m\}$ is equal to $\{C_1, \ldots, C_n\}$. □

Our algorithm considers exactly the same set of candidate clauses and definitions as the one that learns over each stable instance. We further prove that this set is intact during generalization. Similar to above, assume that *Generalize*(C, I, e', Σ) denotes the clause generated by generalizing clause C to cover example e' over (heterogeneous) database I with the set of MDs Σ. Also, let $\mathbf{C}$ be a set of stable clauses and let *Generalize*$_s$ $(\mathbf{C}$, *StableInstances* $(I, \Sigma), e')$ be the set of stable clauses generated by generalizing every stable clause in $\mathbf{C}$ to cover example e' over every stable instance of I.

Theorem 5.4.11. *Given database I with MDs Σ and set of examples E*

$$\begin{aligned}&\mathit{Generalize}_s(\mathbf{C}, \mathit{StableInstances}(I, \Sigma), e') = \\ &\qquad \mathit{StableDefinitions}(\mathit{Generalize}(C, I, e', \Sigma))\end{aligned}$$

Proof. Let $\mathbf{J} = \textit{StableInstances}(I, \Sigma)$. We show that the definition $\textit{Generalize}(C, I, e', \Sigma)$ $= C^*$ is a compact representation of $\textit{Generalize}_s(\mathbf{C}, \mathbf{J}, e') = \{C_1^*, \ldots, C_n^*\}$, i.e. $\textit{StableDefinitions}(C^*) = \{C_1^*, \ldots, C_n^*\}$.

Assume that the schema is $\mathcal{R} = \{R_1(A, B),\ R_2(B, C)\}$ and we have MD $\phi : R_1[B] \approx R_2[B] \rightarrow R_1[B] \rightleftharpoons R_2[B]$. This proof generalizes to more complex schemas. Assume that database instance I contains tuples $R_1(a, b)$, $R_2(b', c)$, and $R_2(b'', c)$, and that $b \approx b'$ and $b \approx b''$. Then, bottom-clause C has the form

$$
\begin{aligned}
T(u) \leftarrow & L'_1, \ldots, L'_{l-1}, \\
& R_1(a, b), R_2(b', c), m_\phi(b, b'), R_2(b'', d), m_\phi(b, b''), \\
& L'_l, \ldots, L'_n,
\end{aligned}
$$

where L'_k, $1 \leq k \leq n$, is a literal.

Now consider two stable instances generated by $\textit{StableInstances}(I, \Sigma)$: J_1, which contains tuples $R_1(a, v_{b,b'})$, $R_2(v_{b,b'}, c)$, $R_2(b'', c)$; and J_2, which contains tuples $R_1(a, v_{b,b''})$, $R_2(b', c)$, $R_2(v_{b,b''}, c)$. The bottom-clause C_1 over instance J_1 has the form

$$
\begin{aligned}
T(u) \leftarrow & L_1, \ldots, L_{l-1}, \\
& R_1(a, v_{b,b'}), R_2(v_{b,b'}, c), R_2(b'', c), \\
& L_l, \ldots, L_n,
\end{aligned}
$$

and the bottom-clause C_2 over instance J_2 has the form

$$
\begin{aligned}
T(u) \leftarrow & L_1, \ldots, L_{l-1}, \\
& R_1(a, v_{b,b''}), R_2(b', c), R_2(v_{b,b''}, c), \\
& L_l, \ldots, L_n,
\end{aligned}
$$

where L_k, $1 \leq k \leq n$, is a literal.

We want to generalize C_1 to cover another training example e'. Let $G_{e'}$ be the ground bottom-clause for e' and $G'_{e'}$ be a stable clause of $G_{e'}$. The literals in C_1 that are blocking will depend on the content of the ground bottom-clause $G'_{e'}$. Assume that the sets of literals $\{L'_1, \ldots, L'_n\}$ in clause C and the set of literals $\{L_1, \ldots, L_n\}$ in clauses C_1 and C_2 are equal. We consider the following cases for the literals that are not equal. The

same cases apply when we want to generalize any other clause generated from a stable instance, e.g., C_2.

Case 1: $G'_{e'}$ contains the literals $R_1(a, v_{b,b'})$ and $R_2(v_{b,b'}, c)$. In this case, $R_1(a, v_{b,b'})$ and $R_2(v_{b,b'}, c)$ are *not* blocking literals, i.e., they are not removed from C_1. $G_{e'}$ also contains literals $R_1(a, b), R_2(b', c), m_\phi(b, b')$. Therefore, the same literals are *not* blocking literals in C either.

Case 2: $G'_{e'}$ contains literals with same relation names but not the same pattern. Assume that $G'_{e'}$ contains the literals $R_1(a, b)$ and $R_2(d, c)$, i.e., they do not join. In this case, literal $R_2(v_{b,b'}, c)$ in C_1 is a blocking literal because it joins with a literal that appears previously in the clause, $R_1(a, v_{b,b'})$. Hence, it is removed. $G_{e'}$ also contains literals $R_1(a, b)$ and $R_2(d, c)$. Because in clause $G'_{e'}$, created from the stable instance, these literals do not join, in $G_{e'}$ they do not join either. In this case, the blocking literal in C is $m_\phi(b, b')$. After literal $m_\phi(b, b')$ is removed, literal $R_2(b', c)$ is not head-connected. Therefore, it is also removed.

Case 3: $G'_{e'}$ contains $R_1(a, v_{b,b'})$, but not $R_2(v_{b,b'}, c)$. In this case, literal $R_2(v_{b,b'}, c)$ is a blocking literal in C_1. Therefore, it is removed. $G_{e'}$ also contains literals $R_1(a, b)$ and $m_\phi(b, b')$, but not $R_2(b', c)$. Therefore, literal $R_2(b', c)$ in C is also blocking and it is removed.

Case 4: $G'_{e'}$ contains $R_2(v_{b,b'}, c)$, but not $R_1(a, v_{b,b'})$. This case is similar to the previous case.

Case 5: $G'_{e'}$ contains neither $R_1(a, v_{b,b'})$ nor $R_2(v_{b,b'}, c)$. In this case, both $R_1(a, v_{b,b'})$ and $R_2(v_{b,b'}, c)$ are blocking; hence they are removed. $G_{e'}$ does not contain literals $R_1(a, b)$, $R_2(b', c)$, $m_\phi(b, b')$. Hence, these literals are also blocking literals in C and are removed.

The generalization operations $\mathit{Generalize}(C, I, e', \Sigma)$ and $\mathit{Generalize}_s(\mathbf{C}, \mathbf{J}, e')$ consist of removing blocking literals from C and $\mathbf{C}$ respectively. We have shown that the same literals are blocking over both the clauses. Therefore, $\mathit{StableDefinitions}(C^*) = \{C_1^*, \ldots, C_n^*\}$. □

This property indicates that CastorX leverages all information available in the heterogeneous database and learns all interesting patterns in its stable instances. Of course, as we follow a more restrictive semantic to provide more accurate definitions overall, our output may *not* be exactly as the one by the method that learns over each stable

instance.

5.4.4 Using the Learned Definitions

The final goal of learning a definition for a target relation is to make predictions. In traditional relational learning, one can make predictions by running the learned Horn definition over the underlying database [46, 50, 60, 67]. Performing prediction and inference is slightly different in our case as it must be done over a database that is *not* stable. Given a learned definition, one has to compute the subset of stable instances that contain the relations in the learned definition. This subset may be significantly smaller than the set of all stable solutions for the entire database. One may follow the semantic of certain answers and compute the final result of the definition as the intersection of its results on all stable instances [6, 12, 13]. Thus, the learned definition guides the user and reduces her effort needed for integration as the alternative approach is to compute all stable instances to learn an accurate definition. In particular, one may be able to learn an accurate definition using the relations of one database without any need to use the ones of other databases. The user can use those relations without any pre-processing effort for prediction. In this case, our approach helps users to avoid spending time and resources on cleaning and computing stable instances for learning and prediction. Users may also avoid computing stable instances by using methods to evaluate approximate answers to a query over the original and non-stable databases [12, 13]. Finally, one may also use the method proposed in Section 5.4.2 for computing coverage to determine whether the learned definition covers an input tuple by generating the ground bottom-clause for the input tuple and checking θ-subsumption for the produced ground bottom-clause and the learned definition. One can also apply the latter to determine the coverage of the input tuple if the learned definition is *not* stable.

5.5 Implementation Details

CastorX is an extension of the Castor relational learning system (Chapter 3). CastorX is implemented on top of a main-memory relational database management system. In this section, we explain implementation details of CastorX.

5.5.0.1 Approximate Coverage Testing

As mentioned in Section 5.4.2, CastorX uses θ-subsumption to compute the coverage of candidate clauses. Ideally, a ground bottom-clause G_e for example e must contain one literal per each tuple in the database that is connected to e through some (similarity) joins. Otherwise, the θ-subsumption test may declare that C does *not* cover e when C actually covers e. However, it is expensive to check θ-subsumption for large clauses. Therefore, to improve efficiency, we use sampling to build ground bottom-clauses. Because we use a sampled bottom-clause G_e^s, checking whether $C \subseteq_\theta G_e^s$ is an approximation of checking whether $I \wedge C \models e$. Our empirical results in Section 5.6.4 indicate that this issue does *not* significantly affect the accuracy of the learned definitions. The learning algorithm involves many (thousands) coverage tests. Because CastorX reuses ground bottom-clauses, it can run efficiently over large databases.

5.5.1 Matching Dependencies

To implement matching dependencies, CastorX uses the similarity operator defined as the average of the *Smith-Waterman-Gotoh* and the *Length* similarity functions. The *Smith-Waterman-Gotoh* function [35] 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. Using a combination of these similarity functions results in a similarity operator that considers two strings as similar if they have local sequence alignments and they have a similar length. Given a string s_1, CastorX considers a string s_2 similar to s_1 if their similarity score is greater than or equal to 0.65 and s_2 is within the top-k_m similar strings to s_1. In our empirical evaluation in Section 5.6, we vary the value of k_m. To improve efficiency, we pre-compute the pairs of similar values according to the similarity operator. We use the pre-computed pairs as an index for similarity search.

5.6 Experiments

We empirically evaluate CastorX to answer the following questions:

1. Can CastorX learn over heterogeneous databases effectively and efficiently? (Sec-

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		
JMDB	43	9.2M	1000	2000
BOM	8	92K		

Table 5.3: Numbers of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset.

tion 5.6.2)

2. What is the benefit of using matching dependencies during learning? (Section 5.6.2)

3. How does the number of training examples affect CastorX's effectiveness and efficiency? (Section 5.6.3)

4. How does sampling affect CastorX's effectiveness and efficiency? (Section 5.6.4)

5.6.1 Experimental Settings

5.6.1.1 Datasets

We use four pairs of databases whose statistics are shown in Table 5.3.

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 [20]. We learn the target relation *dramaRestrictedMovies(imdbId)*, which contains the *imdbId* of movies that are of *drama* genre and 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. We specify the MD

$$IMDB.movies[title] \approx OMDB.movies[title] \rightarrow \\ IMDB.movies[title] \rightleftharpoons OMDB.movies[title].$$

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)**. The original dataset in the Magellan data repository [20] contains ground truth that matches IMDB and OMDB movies. We use the ground truth to generate positive examples. To generate negative examples, we generate movies that are not of *drama* genre or rated R and sample them to obtain twice the number of positive examples.

2. **Walmart + Amazon**: The Walmart and Amazon databases contain information about products, such as their brand, price, categories, dimensions, and weight [20]. We learn the target relation *upcOfComputersAccessories(upc)*, which contains the *upc* of products that are of category *Computers Accessories*. The *upc* is contained in the Walmart database and the information about categories of products is contained in the Amazon database. We specify the MD

$$Walmart.products[title] \approx Amazon.products[title] \rightarrow \\ Walmart.products[title] \rightleftharpoons Amazon.products[title].$$

The original dataset in the Magellan data repository [20] contains ground truth that matches Walmart and Amazon products. We use the ground truth to generate positive examples. To generate negative examples, we generate products that are not computer accessories and sample them to obtain twice the number of positive examples.

3. **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 [20]. The information in the Google Scholar database is not clean, complete, or consistent, e.g., many tuples are missing the year of publica-

tion. 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 specify the MDs

$$DBLP.papers[title] \approx GoogleScholar.papers[title] \rightarrow \\ DBLP.papers[title] \rightleftharpoons GoogleScholar.papers[title] \\ DBLP.papers[venue] \approx GoogleScholar.papers[venue] \rightarrow \\ DBLP.papers[venue] \rightleftharpoons GoogleScholar.papers[venue].$$

The original dataset in the Magellan data repository [20] contains ground truth that matches DBLP and Google Scholar papers. We use the ground truth to generate positive examples. To generate negative examples, we generate pairs of Google Scholar ids and years of publication that are not correct, and sample them to obtain twice the number of positive examples.

4. **JMDB + BOM**: We scrape the Box Office Mojo (BOM) website to obtain a list of movies and their total grossing. We use the JMDB database (*jmdb.de*), which contains information from the IMDb website in relational format. We use these databases to learn a definition for the target relation *highGrossing(title)*, which indicates that the movie with title *title* is high grossing. From BOM, we obtain the top 1K grossing movies and use them as positive examples, and obtain the lowest 2K grossing movies and use them as negative examples. We specify the MD

$$BOM.highGrossing[title] \approx JMDB.movies[title] \rightarrow \\ BOM.highGrossing[title] \rightleftharpoons JMDB.movies[title]$$

Notice that the learning time of relational learning algorithms is not only affected by the number of training examples, but also by the number of tuples in the underlying database. CastorX builds bottom-clauses for training examples. Learning over a large database often results in large bottom-clauses, which translates to long learning times. Therefore, learning over relational databases is generally time-consuming [54, 67]. In Section 5.6.3 we evaluate the effect of the number of training examples on CastorX.

5.6.1.2 Systems and Environment

We compare CastorX against three baseline systems.

1. **Castor-NoMDs**: We use the relational learning system Castor [54] to learn over the original databases. Castor does *not* use MDs. We use Castor because it scales to large databases.

2. **Castor-Exact**: We use Castor [54], but allow the attributes that appear in an MD to be joined through exact joins. Therefore, this system uses information from MDs but only considers exact matches between values.

3. **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 CastorX. Once the entities are resolved and the databases are integrated, we use Castor [54] to learn over the unified database.

4. **CastorX**: We use CastorX, as described in Section 5.4.

CastorX uses the parameter *sample size* to restrict the size of (ground) bottom-clauses. We fix *sample size* to 10. In Section 5.6.4, we evaluate the impact of this parameter on CastorX's effectiveness and efficiency.

We refer to the quality of a definition as the *effectiveness* of the definition. We use the F1-score to measure the effectiveness of definitions. The *F1-score*, which is the harmonic average of the precision and recall, measures the effectiveness of the learned definitions. We use the learning time to measure the *efficiency* of all systems. We perform 5-fold cross validation over all datasets. We evaluate the F1-score and learning time, showing the average over the cross validation.

All systems use 16 threads to parallelize coverage testing. All experiments were run on a server with 30 2.3GHz Intel Xeon E5-2670 processors, running CentOS Linux with 500GB of main memory.

5.6.2 Effectiveness and Efficiency

Table 5.4 shows the results of learning over all datasets using CastorX and the baseline systems. Over all datasets, CastorX obtains a better F1-score than the baselines. Cas-

Dataset	Metric	Castor-NoMDs	Castor-Exact	Castor-Clean	CastorX		
					$k_m = 2$	$k_m = 5$	$k_m = 10$
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
JMDB + BOM	F1-score	0	0.01	0.84	**0.85**	0.74	0.58
	Time (m)	0.36	1.03	7.83	10.3	29.73	51.93

Table 5.4: Results of learning over all datasets. Number of top similar matches denoted by k_m.

torX uses information in the input MDs to find relevant information from all databases. Castor-NoMDs does not learn any definition in the DBLP + Google Scholar dataset. Over this dataset, Castor cannot access information from the DBLP database. Therefore, it is not able to find a reasonable definition. We see a similar result in the JMDB + BOM dataset.

Castor-Exact is able to learn a definition over all datasets. However, as it relies on exact matches, the learned definitions are not as effective as the definitions learned by CastorX. In particular, we see that Castor-Exact obtains a low F1-score over the JMDB + BOM dataset. In this dataset, the same entities over these databases are consistently represented by different names. On the other hand, Castor-Exact obtains a competitive F1-score in the IMDB + BOM dataset with three MDs. The MDs that match cast members and writer names between the two databases contain many exact matches. Therefore, Castor-Exact is able to find paths that connect movies in IMDB with movies in OMDB.

Castor-Clean outperforms the other baselines. Therefore, integrating the input databases by using a simple entity resolution technique provides benefits for learning effective definitions. With the correct value for k_m, CastorX outperforms Castor-Clean in all datasets.

CastorX is able to learn effective definitions over heterogeneous databases efficiently. The effectiveness of the definitions learned by CastorX 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, CastorX is able to ignore these false matches during learning. However, when using multiple MDs or when learning a difficult concept, a high k_m value affects CastorX's effectiveness. In these cases, incorrect matches represent noise that affects CastorX's ability to learn an effective definition. CastorX's effectiveness is lower with higher values of k_m in the IMDB + OMDB (three MDs) dataset and in the JMDB + BOM dataset. For this reason, it is *not* always better to use multiple MDs. CastorX obtains an effective definition over the IMDB + OMDB with one MD and is significantly more efficient than using three MDs.

These experiments show that with the right value for k_m, CastorX can learn effective definitions. Therefore, one should find the right value for k_m by experimenting with different values. In general, CastorX performs well with a sufficiently small k_m, with the benefit that it can learn directly from the dirty data. As k_m increases, the learning time also increases. Therefore, there is a trade-off between the looseness of the similarity function used in MDs and CastorX's effectiveness and efficiency.

The alternative approach to learning over heterogeneous databases is to generate all stable instances and then learn over each of them. However, there may exist an extremely large number of stable instances. For instance, the movie *American Splendor - 2003* in IMDB matches 145 movies in the OMDB database. Just from this movie, we would get 145 distinct stable instances. Integrating and cleaning databases would take significant amount of time and manual labor. Materializing all stable instances would take a huge amount of space. Further, learning over each stable instance would be time-consuming. Therefore, the biggest benefit of CastorX is that it avoids these problems by learning directly from dirty data.

The definitions learned by CastorX and Castor are interpretable. Therefore, we can analyze the learned definitions. For instance, over the JMDB + BOM dataset, CastorX learns the following definition:

$$
\begin{aligned}
\mathit{highGrossing}(x) \leftarrow & \mathit{movies}(y, x, z), \mathit{mov2genres}(y, \text{`}\mathit{Fantasy}\text{'}), \\
& \mathit{mov2countries}(y, \text{`}\mathit{USA}\text{'}), \mathit{releaseseason}(y, \text{`}\mathit{Winter}\text{'}), \\
& \mathit{technical}(y, \text{`}\mathit{PFM} : \mathit{35mm}\text{'}).
\end{aligned}
$$

On the other hand, Castor-NoMDs is *not* able to learn any definition. Castor-Exact can only access information in JMDB for movies whose titles match exactly with titles in BOM, and it learns the following definition:

$$highGrossing(x) \leftarrow movies(y, x, z), mov2genres(y, \text{`}Fantasy\text{'}).$$

5.6.3 Scalability of CastorX

One advantage of relational learning algorithms is that they are data-efficient, i.e., they can learn effective definitions from a small number of training examples [26]. However, for a given learning task, it is not clear how many examples are needed. In this section, we evaluate the effect of the number of training examples in both CastorX's effectiveness and efficiency. We use the IMDB + OMDB (three MDs) dataset and fix $k_m = 2$. We generate 2100 positive and 4200 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, 1000, and 2000 positive examples, and double the number of negative examples. For each training set, we use CastorX to learn a definition. Figure 5.1 shows the F1-scores and learning times for each training set. With 100 positive and 200 negative examples, CastorX 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, CastorX is able to learn efficiently even with the largest training set.

5.6.4 Effect of Sampling

In this section, we evaluate the effect of sampling on CastorX'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 1600 negative examples for training, and 200 positive and 400 negative examples for testing. Figures 5.2 and 5.3 show the F1-score and learning time of CastorX 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

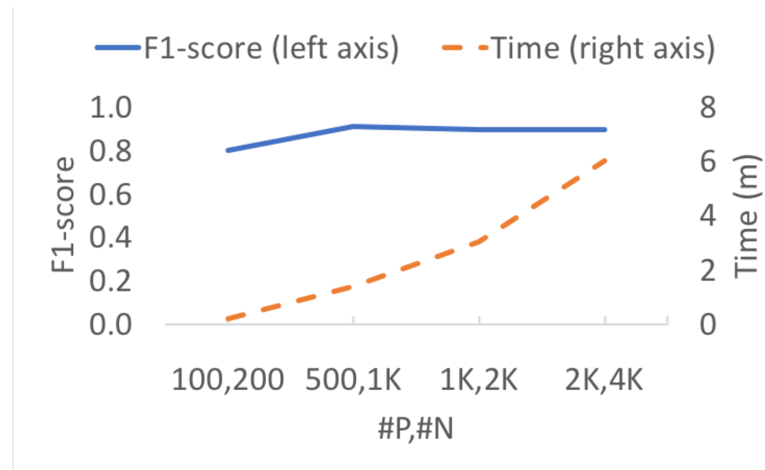

Figure 5.1: Results of learning over the IMDB+OMDB (three MDs) dataset while increasing the number of positive and negative (#P, #N) training examples.

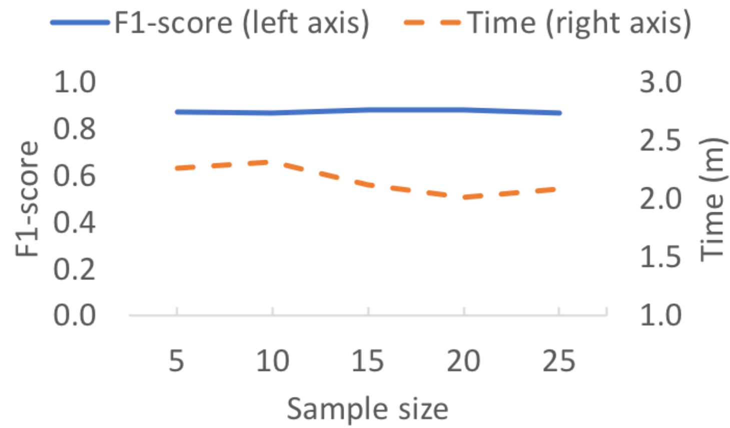

Figure 5.2: Results of learning over the IMDB+OMDB (three MDs) dataset while increasing sample size for $k_m = 2$.

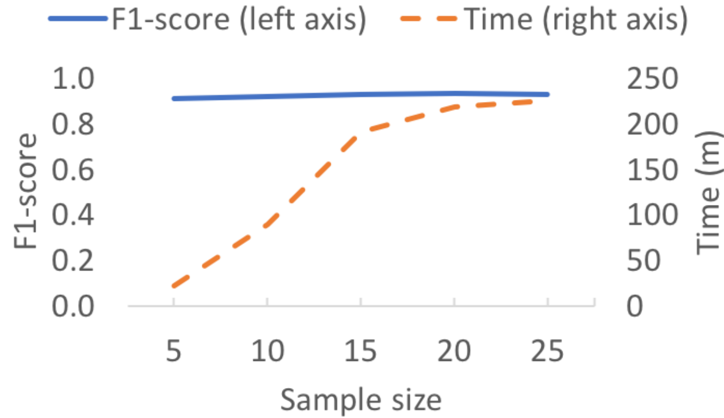

Figure 5.3: Results of learning over the IMDB+OMDB (three MDs) dataset while increasing sample size for $k_m = 5$.

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.

5.7 Related Work

Surveys on relational learning algorithms and their applications have been written by Getoor and Taskar [34] and Muggleton et al. [49]. We have witnessed a surge in building efficient relational learning algorithms for large databases [31, 54, 66, 67]. Our work complements this line of work by creating an efficient relational learning system over heterogeneous databases.

Entity resolution is an active and important area of research in data management whose aim is to find whether two data values or tuples refer to the same real-world entity [33]. One may use these efforts to implement precise similarity operations for MDs.

There have been multiple efforts on modeling value and entity heterogeneity in databases using declarative constraints and defining the properties of clean instances under them [5, 9, 11, 12, 13, 30, 32, 39, 38, 64]. Our effort extends the same approach of dealing with heterogeneous data by efficiently learning over the original database.

A different approach to querying and analyzing multiple databases is to use the avail-

able constraints, including MDs, define concrete similarity and matching functions for the databases, and create and materialize a unified database instance [17, 23, 59]. One may represent the inherent uncertainties in similarities between producing matched values as probabilities and store the final integrated database in a probabilistic database [62]. This approach is preferable if one is able to define a similarity function that accurately quantifies the degree of similarity between values in the domain and a matching function that is able to generate the correct unified value from its input. Such a matching function is hard and resource-intensive to define and usually requires a great deal of expert attention and manual labor. Also, one is *not* usually able to define a general one that can be used across various domains. Our goal is to create a learning algorithm that can be used over databases from various domains. Moreover, if the input databases are relatively large, the materialized database has to accommodate tuples from numerous stable instance causing the database to become extremely large. Learning over such a database is challenging. Our proposed method is able to learn over the original database, which contains significantly fewer tuples and avoids this challenge.

ActiveClean gradually cleans a dirty dataset to learn a convex-loss model, such as Logistic Regression [42]. Its goal is to clean the underlying dataset such that the learned model becomes more effective as it receives more cleaned records. Cleaning may be done manually by an expert. Our objective, however, is to learn a model over dirty data without cleaning and transforming it. Furthermore, ActiveClean does *not* address the problem of having multiple cleaned versions of the database and assumes that the dataset has a unique clean instance.

Chapter 6: Conclusion and Future Work

6.1 Summary

The thesis of this work is that it is possible to develop robust learning algorithms that learn in the presence of representational variations by using data dependencies. Data dependencies contain useful information about the structure and content of the data. By exploiting this information, learning algorithms can be robust against several types of representational variations, such as structural heterogeneities and content heterogeneities.

Chapter 3 introduces the concept of schema independence. An algorithm that is schema independent is robust against structural heterogeneities. We developed Castor, a system that achieves schema independence by using inclusion dependencies. Further, Castor is able to learn concepts over large databases efficiently.

Generally, relational learning algorithms do not scale to large databases. Chapter 4 introduces systems and techniques that allow relational learning algorithms such as Castor to learn efficiently over large databases. We developed AutoMode, a system that automatically generates the language bias used by common relational learning algorithms. Further, we proposed sampling techniques that are integrated into the candidate generation algorithms in Castor. Using sampling allows Castor to learn over large databases efficiently and effectively.

Finally, Chapter 5 considers the problem of learning directly over heterogeneous data. We leveraged the concept of declarative constraint for data cleaning and defined a novel semantic for learning over such databases. We developed CastorX, a system that learns over unclean data. CastorX uses matching dependencies, which can be easily provided by the user, to find the relevant information across databases and learn effective definitions.

6.2 Future Work

We believe that this work initiates some exciting new investigations on the development of robust learning algorithms. Data heterogeneities are ubiquitous. At the rate in which new data is generated, it becomes increasingly difficult to clean and transform the data. We propose a new approach that builds upon the idea of on-demand data preparation for learning algorithms. Users provide only a set of declarative (logical) constraints about the schema or content of the database. We equip the learning algorithms with techniques that use these constraints to implicitly explore data transformations, automatically determine which data is useful, and learn directly from dirty data. This approach removes the need of transforming the data before applying learning algorithms.

We see two exciting future directions for this research. One interesting direction is to explore how this approach extends to other types of learning algorithms. In this work, we focused on relational learning algorithms, which learn from structured data. There exist numerous learning algorithms that learn different types of models over different types of data. The most popular and commonly-used learning algorithms assume that data is stored in a single table. It would be interesting to explore how to make this type of algorithms robust to representational variations. There are already efforts to develop learning algorithms that follow an on-demand data preparation approach [42]. However, these approaches still require significant manual work from the user. We envision algorithms that, if equipped with the proper data constraints, can learn over heterogeneous data without any user involvement.

Another interesting future direction is to explore several types of representational variations and how to encode the transformations that resolve these variations into data constraints. In this work, we focused on popular declarative data constraints such as inclusion dependencies and matching dependencies. These constraints are normally used for designing schemas that guarantee data integrity, as well as for doing data cleaning. However, we believe that the proposed approach can be extended to support other types of transformations and data constraints. On one hand, it would be interesting to explore different types of schema transformations. For example, several transformations have been studied for data exchange and integration [28]. On another hand, it would be interesting to explore different types of content transformations and data cleaning operations. Multiple types of declarative constraints are currently being used for data

cleaning [5, 11, 13, 32, 59, 64]. It would be interesting to explore how to cast this constraints into operations that can transform data automatically and integrate them into machine learning algorithms.

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] Dana Angluin, Michael Frazier, and Leonard Pitt. Learning conjunctions of Horn clauses. *Machine Learning*, 9:147–164, 1992.

[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] Marta Arias, Roni Khardon, and Jérôme Maloberti. Learning Horn expressions with LogAn-H. *Journal of Machine Learning Research*, 8:549–587, 2007.

[8] Paolo Atzeni, Giorgio Ausiello, Carlo Batini, and Marina Moscarini. Inclusion and Equivalent Between Relational Database Schemata. *TCS*, 1982.

[9] 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.

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

[11] 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.

[12] 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.

[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] Balder Ten Cate, Víctor Dalmau, and Phokion G. Kolaitis. Learning schema mappings. *ACM Trans. Database Syst.*, 38:28:1–28:31, 2013.

[15] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On random sampling over joins. In *SIGMOD Conference*, 1999.

[16] Yang Chen, Sean Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. Ontological pathfinding: Mining first-order knowledge from large knowledge bases. In *SIGMOD Conference*, 2016.

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

[18] Edgar F. Codd. Does your DBMS run by the rules? *ComputerWorld*, 1985.

[19] Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4:465–491, 2003.

[20] 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`.

[21] Jesse Davis, Elizabeth S. Burnside, Inês de Castro Dutra, David Page, Raghu Ramakrishnan, Vítor Santos Costa, and Jude W. Shavlik. View learning for statistical relational learning: With an application to mammography. In *IJCAI*, 2005.

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

[23] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. In *CIDR*, 2017.

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

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

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

[27] Ronald Fagin. Inverting schema mappings. *ACM Trans. Database Syst.*, 32:25, 2006.

[28] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[29] Wenfei Fan and Philip Bohannon. Information Preserving XML Schema Embedding. *ACM Trans. Database Syst.*, 33(1):4:1–4:44, 2008.

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

[31] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *The VLDB Journal*, 24:707–730, 2015.

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

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

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

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

[36] Alon Y. Halevy, Michael J. Franklin, and David Maier. Principles of dataspace systems. In *PODS*, 2006.

[37] Joseph M. Hellerstein. People, computers, and the hot mess of real data. In *KDD*, 2016.

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

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

[40] Richard Hull. Relative Information Capacity of Simple Relational Database Schemata. In *PODS*, 1984.

[41] Roni Khardon. Learning function-free Horn expressions. *Machine Learning*, 37:241–275, 1999.

[42] 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.

[43] Ondrej Kuzelka and Filip Zelezný. A restarted strategy for efficient subsumption testing. *Fundam. Inform.*, 89:95–109, 2008.

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

[45] Marcin Malec, Tushar Khot, James Nagy, Erik Blasch, and Sriraam Natarajan. Inductive logic programming meets relational databases: An application to statistical relational learning. In *ILP*, 2016.

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

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

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

[49] Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20. *Machine Learning*, 86:3–23, 2011.

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

[51] Frank Olken. *Random Sampling from Databases.* PhD thesis, UC Berkeley, 1993.

[52] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. *PVLDB*, 8:774–785, 2015.

[53] Jose Picado, Sudhanshu Pathak, Arash Termehchy, and Alan Fern. AutoMode: Relational learning with less black magic. *CoRR*, 2017.

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

[55] Jose Picado, Arash Termehchy, Alan Fern, and Parisa Ataei. Logical scalability and efficiency of relational learning algorithms. *The VLDB Journal*, 28(2):147–171, 2019.

[56] Jose Picado, Arash Termehchy, and Sudhanshu Pathak. Learning efficiently over heterogeneous databases. *PVLDB*, 11:2066–2069, 2018.

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

[58] Chandra Reddy and Prasad Tadepalli. Learning Horn definitions: Theory and an application to planning. *New Generation Computing*, 17:77–98, 1999.

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

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

[61] Ashwin Srinivasan. *The Aleph Manual*, 2004.

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

[63] Arash Termehchy, Marianne Winslett, and Yodsawalai Chodpathumwan. How schema independent are schema free query interfaces? *ICDE*, pages 649–660, 2011.

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

[65] Zhepeng Yan, Nan Zheng, Zachary G. Ives, Partha Pratim Talukdar, and Cong Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *PVLDB*, 6:205–216, 2013.

[66] Xiaoxin Yin, Jiawei Han, Jiong Yang, and Philip S. Yu. Crossmine: efficient classification across multiple database relations. *ICDE*, pages 399–410, 2004.

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