Schema Independent Relational Learning

Jose Picado Arash Termehchy Alan Fern Parisa Ataei
School of EECS, Oregon State University
{picadolj,termehca,alan.fern,ataeip}@oregonstate.edu

ABSTRACT

Learning novel relations from relational databases is an important problem with many applications. Relational learning algorithms learn the definition of a new relation in terms of existing relations in the database. Nevertheless, the same database may be represented under different schemas for various reasons, such as data quality, efficiency and usability. The output of current relational learning algorithms tends to vary quite substantially over the choice of schema. This variation complicates their off-the-shelf application. We introduce and formalize the property of schema independence of 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 that current algorithms are not schema independent. We propose Castor, a relational learning algorithm that achieves schema independence by leveraging data dependencies.

1. INTRODUCTION

Over the last decade, users' information needs over relational databases have expanded from answering precise queries to using machine learning in order to discover interesting and novel relations and concepts [3, 12, 5]. For instance, consider the UW-CSE database [23], which contains information about an academic department. Given this database, we may want to predict the *advisedBy(stud,prof)* relation, which indicates that student *stud* is advised by professor *prof*. Machine learning algorithms often assume that data is represented in a single table. The contents of the table are the features that capture the essential information required to predict the target relation, i.e., *advisedBy*. In a typical scenario, we would be required to hand-engineer this fixed set of features [3]. Each feature would be the result of a query to the database. We would then compute the features for each example in the training data, and store the resulting feature vectors in the table. Finally, we would run a learning algorithm.

Three challenges arise with the described approach. First, hand-engineering features is not an easy task. It is a slow and tedious process and requires significant expertise [3]. It also restricts the algorithm from identifying patterns that are not reflected in the features or combinations of features. Second, by condensing information into a vector of features, we may lose the relational structure, which translates into loss of information. Third, the result of the algorithm may be hard to interpret by users.

In contrast to "table-based approaches", relational machine learning (also called relational learning) attempts to learn concepts directly from a relational database. Given a database and training instances of a new target relation, relational learning algorithms attempt to induce (approximate) relational definitions of the target in terms of existing relations [17, 21, 28]. For example, given an instance of the Original schema for the UW-CSE database in Table 1, the goal may be to induce a definition of the missing relation *advisedBy(stud,prof)* based on a training set of known student-advisor pairs. Learned definitions are usually first-order formulas, sometimes restricted to Datalog programs. Importantly, such relational learning algorithms do not require the intermediate step of feature engineering. This fact, arguably, allows for the easier deployment of machine learning in the context of relational databases [21, 28].

Since the space of possible definitions (e.g. all Datalog rules) is enormous, relational learning algorithms must employ heuristics, or biases, to search for effective definitions. Unfortunately, such heuristics typically depend on the precise choice of schema of the underlying database, which means that the learning output is schema dependent. This is true even if the schemas represent essentially the same information. As an example, Table 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 [1]. A database designer may use a schema closer to the 4NF schema in Table 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*. This would result in a more understandable schema with shorter query execution time, without introducing any redundancy.

EXAMPLE 1.1. *We use the classic relational learning algorithm FOIL [21] to induce a definition for the relation advisedBy(stud,prof) over the Original and 4NF schemas of the UW-CSE database, shown in Table 1. FOIL learns a Datalog rule by starting from an empty rule and iteratively adding atoms to the rule such that the resulting rule at each step has the best score: it covers the most positive and the fewest negative examples. FOIL learns the following Datalog rule over the UW-CSE database with the Original schema:* $advisedBy(x, y) \leftarrow yearsInProgram(x, 7), publication(z, x), publication(z, y)$*, which covers 5 positive examples and 0 negative examples. Because* $yearsInProgram(x, 7)$ *covers the most posi-*

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035923

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

Table 1: Schemas for the UW-CSE dataset.

tive and the fewest negative examples in this database, FOIL picks it as the first atom and proceeds by adding the rest of atoms to the rule. On the other hand, FOIL learns the following rule over the 4NF schema: $advisedBy(x, y) \leftarrow student(x, post\_generals, 5)$, $professor(y, faculty)$, $publication(z, y)$, $taughtBy(v, y, w)$*, which covers 12 positive examples and 10 negative examples. In this case, FOIL first selects* $student(x, post\_generals, 5)$*. Because FOIL does not backtrack, then the definitions over both schemas are different, even if the rest of the atoms added to the rules are the same. Intuitively, the definition learned over the Original schema better expresses the relationship between an advisor and advisee.*

Generally, there is no canonical schema for a particular set of content in practice and people often represent the same information in different schemas for several reasons [1, 9]. For example, it is generally easier to enforce integrity constraints over highly normalized schemas [1]. 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 [1]. Thus, a database designer may sacrifice data quality and choose a more *denormalized* schema for its data to achieve better usability and/or performance. She may also hit a middle ground by choosing a style of design for some relations and another style for other relations in the schema. Further, as the relative priorities of these objectives change over time, the schema will also evolve.

Users generally have to restructure their databases, in order to effectively use relational learning algorithms, i.e., deliver definitions for the target concepts that a domain expert would judge as correct and relevant. To make matters worse, these algorithms do not normally offer any clear description of their desired schema and database users have to rely on their own expertise and/or do 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 processing [2], schema mapping [5], and entity resolution [11]. 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 [9]. 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 via join and project operations, i.e. composition and decomposition [1]. However, the number of possible schemas within a particular family of a data set are extremely large. For example, a relational table may have exponential number of distinct decompositions. As many learning algorithms need some time for parameter tuning under a new schema [14], it may take a prohibitively long time to find the best schema. Further, one has to transform the underlying data to the desired schema, which may not be practical for a large and/or constantly evolving database. Another possible approach is to define a *universal schema* to which all possible schemas can be transformed and use or develop algorithms that are effective over this schema. The experience gained from the idea of universal relation indicates such schemas may not always exist [1]. Users also have to transform their databases to the universal schema, which may be quite complex and time-consuming, especially for large and/or constantly evolving databases.

In this paper, we introduce the novel property of *schema independence* for relational learning algorithms, i.e., the ability to deliver the same answers regardless of the choices of schema for the same data. 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 paper are:

- We introduce and formally define the property of schema independence (Section 3), which formalizes the notion of a learning algorithm returning equivalent answers over schema transformations that preserve information content.
- We analyze the property of schema independence for the families of top-down [17, 21] and bottom-up [18, 19] relational learning algorithms. We show that top-down algorithms are not schema independent under (de) composition transformations (Section 5). We formally analyze ProGolem [19], a bottom-up algorithm, and show that it is not schema dependent under (de) composition (Section 6).
- We introduce Castor, a bottom-up algorithm that is schema independent under vertical (de) composition (Section 7). Castor achieves schema independence by integrating database constraints into the learning algorithm. Castor uses various techniques to learn efficiently over large databases.
- 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 two real-world databases (Section 8). Our empirical results generally confirm our theoretical results.

Further details on our work and proofs for our theoretical results can be found in [20].

2. BACKGROUND

2.1 Related Work

There has been a growing interest in developing relational learning algorithms that scale to large databases [28, 27, 10]. QuickFOIL [28] provides an in-RDBMS implementation of a modified version of FOIL. AMIE+ [10] learns rules from RDF-style knowledge bases, which contain binary relations. These systems focus on scaling learning algorithms for large databases. We develop a schema independent relational learning algorithm and, as opposed to them, do not modify the internals of the RDBMS. The system in [15] learns linear models over multiple relations efficiently. Our aim, however, is to achieve schema independence. Also, their system assumes that all relations can be joined into one universal relation, which is not generally true in relational databases. Moreover, it learns linear models and not Datalog definitions. Researchers have noticed that using likelihood functions to measure joint probability distributions of attributes in a relational database may lead to different results over certain variations of the database design [24]. They have proposed using pseudolikelihood functions to approximate the joint probability distributions, which is robust over some schema variations. Nevertheless, the authors in [24] do not provide any general and formal framework to explore the sensitivity of learning algorithms to the database design.

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 [13, 9]. Another notable group of database transformations is schema mapping for data exchange [8]. These transformations may 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 [7]. Nevertheless, these mappings may introduce labeled nulls to the database instance. To the best of our knowledge, relational learning over database instances with labeled nulls has not been precisely defined and explored. It takes more space than a single paper to include investigations of relational learning over databases with labeled null and define schema independence property for transformations that introduce labeled nulls and we leave them as interesting future work. Researchers have defined the property of design independence for keyword query processing over XML [26]. We extend this line of work by formally exploring the property of schema independence for relational learning algorithms.

In this paper, we extend the principle of logical data independence [1] for relational learning algorithms. The property of schema independence differs with the idea of logical data independence in an important issue. One may achieve logical data independence by an affordable amount of experts' intervention, 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 expert's intervention.

2.2 Basic Definitions

We fix two disjoint (countably) infinite sets of relation 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. *Schema* $\mathcal{R}$ is a pair $(\mathbf{R}, \Sigma)$, where $\mathbf{R}$ and Σ are finite sets of relations 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. Let $\pi_X(I_R)$, $X \subseteq sort(R)$, denote the projection of relation I_R 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 $R[X] \subseteq S[X]$ and $S[X] \subseteq R[X]$ hold in a schema, we denote them as $R[X] = S[X]$ 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., FD $X \rightarrow Y$ and $Y \rightarrow Z$ imply $X \rightarrow Z$ [1]. The set of all constraints implied by Σ is shown as 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_is 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 *definite Horn clause* (Horn clause for short) is a finite set of literals that contains 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 clause has the form: $T(\mathbf{u}) \leftarrow L_1(\mathbf{u_1}), \cdots, L_n(\mathbf{u_n})$. Horn clauses are also called conjunctive queries [1]. A *Horn definition*, i.e., union of conjunctive queries, is a set of Horn clauses with the same head literal. A Horn definition is defined over a schema if the body of all clauses in the definition contain only literals whose relations are in the schema. In this paper, we use Horn definitions to define new target relations that are not in the schema. Thus, the heads of all clauses in these definitions are the *target relation*.

3. FRAMEWORK

3.1 Relational Learning

Relational learning can be viewed as a search problem for a hypothesis that deduces the training data, following either a top-down or bottom-up approach. Top-down algorithms [21, 17] start from the most general hypothesis and employ specialization operators to get more specific hypotheses. A common specialization operator is the addition of a new literal to the body of a clause. On the other hand, bottom-up algorithms [18, 19] start from specific hypotheses that are constructed based on ground training examples, and use generalization operators to search the hypothesis space. Generalization operators include inverse resolution, relative least general generalization, asymmetric relative minimal generalization, among others. Thus, a relational learning algorithm is a sequence of steps, where in each step an operator is applied to the current hypothesis.

Inductive Logic Programming (ILP) is the subfield of machine learning that performs relational learning by learning first-order definitions from examples and an input relational database. In this paper we use the names ILP algorithm and relational learning algorithm interchangeably. Training examples E are usually tuples of a single target relation T, which express positive (E^+) or negative (E^-) examples. The learned definitions are called the hypothesis H, which is usually restricted to Horn definitions for efficiency reasons. Given a database instance I, positive examples E^+, negative examples E^-, and a target relation T, the task of a relational learning algorithm is to find a definition H for T such that $\forall p \in E^+, H \wedge I \models p$ (completeness) and $\forall p \in E^-, H \wedge I \not\models p$ (consistency). The input database instance I is also called *background knowledge*. In the following sections we provide concrete definitions of several relational learning algorithms.

EXAMPLE 3.1. *Consider using a relational learning algorithm and the UW-CSE database with the Original schema shown in Table 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 definition $collaborated(x, y) \leftarrow publication(p, x), publication(p, y)$. This is a complete and consistent definition with respect to the training data, and indicates that two persons have collaborated if they are co-authors.*

In this paper, we study relational learning algorithms for Horn definitions. 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 (e.g. time and space) to explore all definitions. 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 $\theta = \langle \theta_1, ..., \theta_r \rangle$, where each θ_i is a parameter for the algorithm. We denote the parameter space by Θ, and it contains all possible parameters for an algorithm. We denote the hypothesis space (or language) of algorithm A over schema $\mathcal{R}$ with parameters θ as $\mathcal{L}^A_{\mathcal{R},\theta}$. Note that not all parameters affect the hypothesis space. For instance, a parameter setting the search strategy to greedy impacts how the hypothesis space is explored, but does not restrict the hypothesis space. The hypothesis space $\mathcal{L}^A_{\mathcal{R},\theta}$ is a subset of $\mathcal{HD}_\mathcal{R}$ [17, 21], and each member of $\mathcal{L}^A_{\mathcal{R},\theta}$ is a hypothesis.

There is a trade-off between computational resources used by an algorithm and the size of its hypothesis space. The hypothesis space is restricted so that the algorithm can be used in practice, with the hope that it finds a consistent and complete hypothesis.

EXAMPLE 3.2. *Continuing Example 3.1, 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 complete and consistent definition* $collaboratedProf(x, y) \leftarrow professor(x),\ professor(y),\ publication(p, x),\ publication(p, y)$. *However, if we set clause-length* $= 3$, *the previous definitions is not in the hypothesis space of the algorithm. Thus, the algorithm cannot learn this or any other complete and consistent definition.*

3.2 Schema Independence

3.2.1 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 [13, 9].

Given schemas $\mathcal{R}$ and $\mathcal{S}$, a *transformation* is a (computable) function $\tau : \mathcal{I}(\mathcal{R}) \rightarrow \mathcal{I}(\mathcal{S})$. For brevity, we write transformation τ as $\tau : \mathcal{R} \rightarrow \mathcal{S}$. Transformation τ is *invertible* iff it is total and there exists a transformation $\tau^{-1} : \mathcal{S} \rightarrow \mathcal{R}$ 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 $I \in \mathcal{I}(\mathcal{R})$. The transformation τ^{-1} may or may not be total. 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 $\tau : \mathcal{R} \rightarrow \mathcal{S}$ 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 [13, 9]. We assume that in order for schemas $\mathcal{R}$ and $\mathcal{S}$ to be information equivalent via τ, τ^{-1} has to be total. Although more restricted, this definition is sufficient to cover the transformations discussed in this paper.

EXAMPLE 3.3. *In addition to the functional dependencies shown in Table 1, let the following inclusion dependencies hold over the relations of Original schema in this table:* $student[stud] = inPhase[stud]$, $student[stud] = yearsInProgram[stud]$, $professor[prof] = hasPosition[prof]$, *One may join relations* $student$, $inPhase$, *and* $yearsInPrograms$ *and join relations* $professor$ *and* $hasPosition$ *to map each instance of the Original schema to an instance of the 4NF schema. Also, each instance of the 4NF schema can be mapped to an instance of the Original schema by projecting relation* $student$ *to relations* $student$, $inPhase$, *and* $yearsInProgram$ *and projecting relation* $professor$ *to relations* $hasPosition$ *and* $professor$. *Hence, these schemas are information equivalent.*

3.2.2 Mapping Definitions

Let $\mathcal{HD}_\mathcal{R}$ be the set of all Horn definitions over schema $\mathcal{R}$. In order to learn semantically equivalent definitions over schemas $\mathcal{R}$ and $\mathcal{S}$, we should make sure that the sets $\mathcal{HD}_\mathcal{R}$ and $\mathcal{HD}_\mathcal{S}$ are equivalent. That is, for every definition $h_\mathcal{R} \in \mathcal{HD}_\mathcal{R}$, there is a semantically 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 semantically 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 $\mathcal{I}(\mathcal{R})$. $h_\mathcal{R}(I)$ shows the result of $h_\mathcal{R}$ on I.

DEFINITION 3.4. *Transformation* $\tau : \mathcal{R} \rightarrow \mathcal{S}$ *is definition preserving w.r.t.* $\mathcal{L}_\mathcal{R}$ *and* $\mathcal{L}_\mathcal{S}$ *iff 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 $\mathcal{S}$. We call function δ_τ a *definition mapping* for τ. Transformation τ is *definition bijective* w.r.t. $\mathcal{L}_\mathcal{R}$ and $\mathcal{L}_\mathcal{S}$ iff τ and τ^{-1} are definition preserving w.r.t. $\mathcal{L}_\mathcal{R}$ and $\mathcal{L}_\mathcal{S}$.

If τ is definition bijective w.r.t. equivalent sets of Horn definitions, one can rewrite each Horn definition over $\mathcal{R}$ as a Horn definition over $\mathcal{S}$ such that they return the same results over all corresponding database instances of $\mathcal{R}$ and $\mathcal{S}$, and vice versa. We call these definitions *equivalent*. We use the operator $\equiv$ to show that two definitions are equivalent.

3.2.3 Relationship Between 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 w.r.t. $\mathcal{HD}_\mathcal{R}$ and $\mathcal{HD}_\mathcal{S}$. If τ is bijective, the learning algorithm takes as input the same background knowledge. Also, a definition bijective transformation ensures that the learning algorithm can output equivalent Horn definitions over both schemas. Nevertheless, it may be hard to check both conditions for given schemas. Next, we extend the results in [9] to find the relationship between the properties of bijective and definition bijective transformations. In this paper, we consider only transformations that can be written as sets of Horn definitions. We call these *Horn transformations*. Composition/ decomposition are well-known examples of Horn transformations [1].

EXAMPLE 3.5. *Let $\mathcal{R}$ be the Original schema and $\mathcal{S}$ be the 4NF schema in Example 3.3. The transformation from the Original schema to the 4NF schema can be written as the following set of Horn definitions:*

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

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

Let transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ and its inverse $\tau^{-1} : \mathcal{S} \rightarrow \mathcal{R}$ be Horn transformations. Clearly, the head of each Horn definition in τ^{-1} will be a relation in $\mathcal{R}$. Let $h_{\mathcal{R}}$ be a Horn definition in $\mathcal{HD}_{\mathcal{R}}$. The composition of $h_{\mathcal{R}}$ and τ^{-1}, denoted by $h_{\mathcal{R}} \circ \tau^{-1}$, is a Horn definition that belongs to $\mathcal{HD}_{\mathcal{S}}$, created by applying $h_{\mathcal{R}}$ to the heads of clauses in τ^{-1} [1]. That is, $h_{\mathcal{R}} \circ \tau^{-1}(J) = h_{\mathcal{R}}(\tau^{-1}(J))$, for all $J \in \mathcal{I}(\mathcal{S})$.

PROPOSITION 3.6. *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 w.r.t $\mathcal{HD}_{\mathcal{R}}$ and $\mathcal{HD}_{\mathcal{S}}$.*

Intuitively, if $\tau : \mathcal{R} \rightarrow \mathcal{S}$ is bijective and both τ and τ^{-1} are Horn transformation, 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 paper, we consider only the bijective Horn transformations whose inverses are Horn transformations.

EXAMPLE 3.7. *Let $\mathcal{R}$ be the Original schema and $\mathcal{S}$ be the 4NF schema in Example 3.3 and $\tau : \mathcal{R} \rightarrow \mathcal{S}$ $\tau^{-1} : \mathcal{S} \rightarrow \mathcal{R}$ are the Horn transformation explained in Example 3.5. According to Proposition 3.6, τ is definition bijective w.r.t. $\mathcal{HD}_{\mathcal{R}}$ and $\mathcal{HD}_{\mathcal{S}}$.*

3.2.4 Schema Independence Property

The **hypothesis space** determines the set of possible Horn definitions that the algorithm can explore. Therefore, the output of a learning algorithm depends on its hypothesis space. In Example 3.2, we 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 semantically 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 Θ be the parameter space for algorithm A.

DEFINITION 3.8. *Algorithm A is hypothesis invariant under transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ iff τ is definition bijective w.r.t. $\mathcal{L}^{A}_{\mathcal{R},\theta}$ and $\mathcal{L}^{A}_{\mathcal{S},\theta}$, for all $\theta \in \Theta$.*

Algorithm A is hypothesis invariant under a set of transformations iff 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. We define a relational learning algorithm as a function $A(I, E, \theta)$ to $\mathcal{L}^{A}_{\mathcal{R},\theta}$. That is, taking as input a database instance I, training examples E, and parameters $\theta \in \Theta$, the algorithm outputs a hypothesis in $\mathcal{L}^{A}_{\mathcal{R},\theta}$.

DEFINITION 3.9. *Algorithm A is schema independent under bijective transformation $\tau : \mathcal{R} \rightarrow \mathcal{S}$ iff A is hypothesis invariant under τ and for every $I \in \mathcal{I}(\mathcal{R})$ and all $\theta \in \Theta$, we have: $A(\tau(I), E, \theta) \equiv \delta_{\tau}(A(I, E, \theta))$, where δ_{τ} is the definition mapping for τ.*

Algorithm A is schema independent under the set of transformations iff 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 3.10. *Consider the Original schema and the 4NF schema in in Example 3.3. 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, 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 1.1, and is not schema independent.*

4. DECOMPOSITION AND COMPOSITION

There are many bijective Horn transformations between relational schemas [13, 1]. It takes more space than a single paper to explore the behavior of relational learning algorithms over all such transformations. In this paper, we explore the schema independence of relational learning algorithms under two widely used Horn transformations called *decomposition*, where the transformation is projection, and *composition*, where the transformation is natural join [1]. Our reasons for selecting these transformations are two fold. First, they are used in most normalizations and denormalizations, e.g., 3rd normal form. which are arguably one of the most frequent schema modifications and their importances have been recognized from the early days of relational model [1]. Database designers often normalize schemas to remove redundancy and insertion/ deletion anomalies and denormalize them to improve query processing time and schema readability [1]. We also observe several cases of them in relational learning benchmarks, one of which is presented in Section 1.

We define decomposition as follows [1]. 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 4.1. *A decomposition of schema $\mathcal{R} = (\mathbf{R}, \Sigma_R)$ with single relation symbol 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*

- *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 $\bowtie_{i=1}^{n} I_{S_i}$ $= I_R$.*
- *For all S_i, S_j, $1 \leq i, j \leq n$, such that $X = sort(S_i) \cap sort(S_j) \neq \emptyset$, Σ_S contains IND with equality $S_i[X] = S_j[X]$.*
- *We have $\Sigma_S = \Sigma_R \cup \lambda$.*

The first and third conditions in Definition 4.1 are generally known as *lossless join* and *dependency preservation* properties, respectively. The second condition in Definition 4.1 ensures 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 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 4.1, e.g., lossless join property, hold in this example due to the FDs in original and 4NF schemas [1]. 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 natural join.

Consider again schema $\mathcal{S}$ in Definition 4.1. The join $\bowtie_{i=1}^{n} I_{S_i}$ is *globally consistent* if for each j, $1 \leq j \leq n$, $\pi_{sort(S_j)} \bowtie_{i=1}^{n} I_{S_i} = I_{S_j}$ [1]. Intuitively speaking, a join is globally consistent if none of its relation has a dangling tuple regarding the join. For example, the join between the relations of $\mathcal{S}$ in the first condition of Definition 4.1 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 [1]. 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 paper, we consider only the decompositions where the join in the first condition of Definition 4.1 is acyclic [1]. Acyclic joins cover most decompositions in real-world [1]. For examples, 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 the decomposition (composition) of a schema with more than one relation as the set of decompositions (compositions) of all its relations. We define a *decomposition/ composition* of a schema as a finite set of applications of composition and/or decomposition to the schema. Every decomposition is bijective [1]. 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. We explore the property of schema independence only for decomposition/ composition in this paper.

5. TOP-DOWN ALGORITHMS

Most relational learning algorithms follow a covering approach [21, 17]. The covering approach consists in constructing one clause at a time. After building a clause, the algorithm adds the clause to the hypothesis, discards 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 $LearnClause$ procedure depends on the nature of the algorithm. In top-down algorithms, the $LearnClause$ procedure in Algorithm 1 searches the hypothesis space from general to specific, by using a refinement (specialization) operator that is generally adding a new literal to the clause.

The hypothesis space in top-down algorithms is a refinement graph, that is a rooted directed acyclic graph in which nodes represent clauses and each arc is the application of a basic refinement operator. The basic strategy of top-down algorithms consists of starting from the most general clause, which corresponds to the root of the refinement graph, and repeatedly refining it until it does not cover any negative example.

The strategy of constructing and searching the refinement graph varies between different top-down algorithms. For instance, FOIL [21, 28] is an efficient and popular top-down algorithm that follows a greedy best-first search strategy. In this section, we analyze the schema independence properties of FOIL. However, the results that we show in this section hold for all top-down algorithms no matter which search strategy they follow.

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

```
Input  : Database instance $I$, positive examples $E^+$, negative
         examples $E^-$
Output: A Horn definition $H$
$H \leftarrow \{\}$ ; $U \leftarrow E^+$
while $U$ is not empty do
    $C \leftarrow LearnClause(I, U, E^-)$
    if $C$ satisfies minimum condition then
        $H \leftarrow H \cup C$
        $U \leftarrow U - \{c \in U | I \cup H \models c\}$
return $H$
```

The refinement graph for most schemas, even the ones with a relatively small number of relations and attributes, may grow significantly [21, 17]. Hence, the construction and search over the refinement graph may become too inefficient to be practical. To be used in practice, FOIL restricts its search space, i.e. hypothesis space. We call the number of literals in a clause its length. A common method is to restrict the maximum length of each clause in the refinement graph [21, 17]. Intuitively, because composition/decompositions modify the number of relations in a schema, equivalent clauses over the original and transformed schemas may have different lengths. Hence, this type of restrictions may result in different hypothesis spaces. One may like to fix this problem by choosing different values for the maximum lengths over the original and transformed schemas. The following theorem proves 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 5.1. *FOIL is not hypothesis invariant.*

Progol is another popular top-down algorithm that follows the same approach as FOIL but considers a larger number of candidate clauses at each step over a more restricted hypothesis space [17]. Theorem 5.1 also applies to Progol. It is confirmed by our experiments in Section 8.

6. BOTTOM-UP ALGORITHMS

Bottom-up algorithms also follow the covering approach shown in Algorithm 1. However, their *LearnClause* procedure searches the hypothesis space from specific to general hypotheses. Given a positive example, bottom-up algorithms attempt to find the most specific clause in the hypothesis space, called *bottom-clause*, that covers the example, relative to the database instance [18, 19]. They generalize these bottom-clauses to find definitions that cover as most positive and as fewest negative examples as possible.

6.1 Bottom-clause Construction

Let $I_{\mathcal{R}}$ be a database instance over schema $\mathcal{R}$. The bottom-clause associated with positive example e, relative to $I_{\mathcal{R}}$, denoted by $\perp_{e, I_{\mathcal{R}}}$, is the most specific clause over $\mathcal{R}$ that covers e, relative to $I_{\mathcal{R}}$. A typical algorithm for computing bottom-clauses using inverse entailment is given in [17]. The algorithm starts with an empty clause, and iteratively adds literals to the clause. Given positive example $T(a_1, \ldots, a_n)$, it assigns a fresh variable u_i to each distinct constant and adds literal $T(u_1, \ldots, u_n)$ to the head of the bottom-clause. The algorithm maintains the mapping between constants and variables. It then finds all tuples in the database that contain constants $a_1, \ldots, a_n$. For each tuple, the algorithm adds a new literal to the bottom-clause, where the predicate symbol is the tuple relation symbol and the terms are variables obtained by replacing $a_1, \ldots, a_n$ in the tuple to their corresponding variables and assigning new variables to newly encountered constants in the tuples. . In the following iterations, the algorithm searches the database for tuples that contain new constants and adds new literals to the bottom-clause. This algorithm may generate very long clauses after multiple iterations over a large database. A common method to restrict the number of iterations is to limit the maximum *depth* of the bottom-clause [17]. The depth of a variable x, denoted by $d(x)$, is 0 if it appears in the head of the clause, otherwise it is $min_{v \in U_x}(d(v)) + 1$, 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 6.1. *This clause over the Original UW-CSE schema in Table 1 has depth 1:* $taLevel(x, y) \leftarrow ta(c, x, t), courseLevel(c, y)$. *The following clause for target relation* $commonLevel(x, y)$*, which says that students x and y assist with courses at the same level 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 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 depth parameter does not result in such equivalent bottom-clauses, because the original and transformed schemas need different depths to create equivalent bottom-clauses.

EXAMPLE 6.2. *Let us compose and replace relations* $courseLevel$ $(crs, level)$ *and* $ta(crs, stud, term)$ *in the Original UW-CSE schema with* $courseLevelTa(crs, level, stud, term)$. $commonLevel$ *from Example 6.1 has the following definition over this schema, which has depth 1:* $commonLevel(x, y) \leftarrow courseLevelTa(c1, l, x, t1),$ $courseLevelTa(c2, l, y, t2)$. *If we set the maximum depth to 1, in the Original schema, the clause in Example 6.1 is not in the hypothesis language. But, under the new schema, the clause presented above is in the hypothesis language.*

Using a similar idea to the proof of Theorem 5.1, the following lemma proves that the bottom-clause construction algorithm is schema dependent even if different depth values are used across schemas.

LEMMA 6.3. *Bottom-clause construction is schema dependent.*

6.2 Generalization

There are multiple bottom-up algorithms whose differences lie mainly in their generalization operator [18, 19, 4]. Most bottom-up algorithms cannot learn efficiently over small or medium databases without making assumptions that do not hold over most real-world databases [19]. ProGolem is a bottom-up algorithm that can run efficiently over small or medium databases without making generally unrealistic assumption [19]. 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's *LearnClause* procedure 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 $\overrightarrow{C}$. ProGolem uses the *asymmetric relative minimal generalization* (*armg*) operator to generalize clauses. 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 of the schema used, such as coverage, which is the number of positive examples minus the number of negative examples covered by the clause.

We now explain the *armg* operator in detail. Let $\perp_{e,I_{\mathcal{R}}}$ be the bottom-clause associated with example e, relative to $I_{\mathcal{R}}$. Let $\overrightarrow{C} = T \leftarrow L_1, \cdots, L_n$ be the ordered version of $\perp_{e,I_{\mathcal{R}}}$. Let e' be another example. L_i is a *blocking atom* iff 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 $I_{\mathcal{R}}$ [19]. Algorithm 3 in Appendix A shows the ARMG algorithm, which implements the *armg* operator. Given the bottom-clause $\perp_{e,I_{\mathcal{R}}}$ and a positive example e', *armg* drops all blocking atoms from the body of $\perp_{e,I_{\mathcal{R}}}$ 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 those literals. For ProGolem to be schema independent, the *armg* operator must return equivalent clauses given equivalent input clauses over original and transformed databases.

EXAMPLE 6.4. *Consider the following equivalent definitions for target relation hardWorking over the Original and 4NF UW-CSE schema in Table 1, respectively:* $hardWorking(x) \leftarrow student(x),$ $inPhase(x, prelim),\ yearsInProgram(x, 3),$ $hardWorking(x) \leftarrow student(x, prelim, 3)$.
Assume that armg wants to generalize these clauses to cover example e'. *Let* e' *satisfy literal* $student(x)$ *but does not satisfy* $inPhase(x, prelim)$. *The armg operator keeps literal* $student(x)$ *in the first clause, but it eliminates* $student(x, prelim, 3)$ *from the second clause. Hence, it delivers non-equivalent generalizations.*

Thus, neither bottom-clause construction nor generalization phases in ProGolem are schema independent.

THEOREM 6.5. *ProGolem is not schema independent.*

Due to schema dependence of the bottom-clause construction algorithm, other bottom-up algorithms are also schema dependent [18].

7. 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 INDs into the bottom-clause construction and generalization algorithms to achieve schema independence. 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 clauses in decomposition/ compositions of $\mathcal{R}$ [1]. For example, consider schema $\mathcal{R}_2$:$\{R_1(A, B), R_2(A, C)\}$ with the IND $R_1[A] = R_2[A]$ and the clause $h_{\mathcal{R}_2} : T(x) \leftarrow R_1(x, y)$. Because each value in $R_1[A]$ also appears in $R_2[A]$, we can rewrite $h_{\mathcal{R}_2}$ as $g_{\mathcal{R}_2} : T(x) \leftarrow R_1(x,y), R_2(x,z)$. Now, consider a composition of $\mathcal{R}$, $\mathcal{S}_2$:$\{S_1(A,B,C)\}$. The clause $h_{\mathcal{S}_2} : T(x) \leftarrow S_1(x,y,z)$ over $\mathcal{S}_2$ is equivalent to both $h_{\mathcal{R}_2}$ and $g_{\mathcal{R}_2}$. $g_{\mathcal{R}_2}$ and $h_{\mathcal{S}_2}$ have also similar syntactic structures: there is a bijection between the distinct variables in $g_{\mathcal{R}_2}$ and $h_{\mathcal{S}_2}$. However, such bijection does not exist between $h_{\mathcal{R}_2}$ and $h_{\mathcal{S}_2}$. As learning algorithms modify the syntactic structure of clauses to learn a target definition and $h_{\mathcal{R}_2}$ and $h_{\mathcal{S}_2}$ 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 $h_{\mathcal{S}_2}$ to generate clause $h^{'}_{\mathcal{S}_2} : T(x) \leftarrow S_1(x,y,x)$. This algorithm cannot apply a similar change to $h_{\mathcal{R}_2}$ as $h_{\mathcal{R}_2}$ does not have any corresponding variable to z. But, the algorithm can apply the same modification to $g_{\mathcal{R}_2}$ and generate an equivalent Horn clause to $h^{'}_{\mathcal{S}_2}$. Moreover, as INDs generally reflect important relationships, it may improve the effectiveness of the algorithm to use them for learning definitions.

Castor's $LearnClause$ procedure is shown in Algorithm 2. It first generates the bottom-clause associated with some positive example using the modified bottom-clause construction algorithm presented in Section 7.1. It minimizes the bottom-clause using the procedure explained in Section 7.4. Then, it performs a beam search to select the best candidate after multiple applications of the modified $ARMG$ algorithm, explained in Section 7.2.1. Finally, it reduces the best candidate using the algorithm explained in Section 7.2.2.

Algorithm 2: Castor's LearnClause algorithm.

```
Input  : Database instance $I$, positive examples $E^+$, negative
         examples $E^-$, parameters $K$ and $N$.
Output: A new clause $C$.
$\overrightarrow{C} \leftarrow Castor\_BottomClause($first example in $E^+)$
$\overrightarrow{C} \leftarrow Minimize(\overrightarrow{C})$ ; $BC \leftarrow \{\overrightarrow{C}\}$
repeat
    $BestScore \leftarrow$ score of highest scoring candidate in $BC$
    $E^+_S \leftarrow K$ randomly selected positive examples from $E^+$
    $NC = \{\}$
    foreach clause $C \in BC$ do
        foreach $e' \in E^+_S$ do
            $C' \leftarrow Castor\_ARMG(C, e')$
            if $Score(C') > BestScore$ then
                $NC \leftarrow NC \cup C'$
    $BC \leftarrow$ highest scoring $N$ candidates from $NC$
until $NC = \{\}$
$C' \leftarrow$ highest scoring candidate in $BC$
Return $Castor\_Reduce(C', I, E^-)$
```

7.1 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 $S_1 \ldots S_n$ in the transformed schema $\mathcal{S}$. 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 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 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 4.1, there are INDs with equality between the join attributes of relation symbols $S_1 \ldots S_n$. We use IND 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 7.1. *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*

- $X_k = sort(S_k) \cap sort(S'_k)$.
- $S_{k+1} = S'_k$ *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 tuples $s_1, \ldots, s_n$ that all participate in the join and none of them is a dangling tuple with the regard to the full join. Otherwise, Castor must check the join condition for each pair of tuples.

EXAMPLE 7.2. *Consider an instance of the original UW-CSE schema in Table 1 with tuples $s_1 : student(Abe)$, $s_2 : inPhase$ $(Abe, prelim)$ and $s_3 : year(Abe, 2)$. 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. 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.*

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

DEFINITION 7.3. *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*

- $S_{i+1} = S'_i$ *for* $1 \leq i \leq n-1$ *and* $S_1 = S'_n$.
- *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 $S_1[B] = S_2[B]$, $S_2[C] = S_3[A]$, 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 4.1 is acyclic, the INDs with equality in the decomposed schema are not cyclic. Thus, Castor does not face the aforementioned issue.

PROPOSITION 7.4. *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 4.1 are not cyclic.*

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 joint 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 7.4 and 8 that using an RDBMS implementation, Castor bottom-clause construction algorithm runs faster than other algorithms.

As explained in Section 6.1, 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 and stops otherwise. Intuitively, since the number of distinct variables in equivalent Horn clauses over composition/ decomposition are equal, this condition helps Castor to return equivalent bottom-clauses over composition/ decomposition. The following Lemma states that Castor bottom-clause construction algorithm is schema independent.

LEMMA 7.5. *Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a composition/ decomposition, 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 $\bot_{e,I} \equiv \bot_{e,\tau(I)}$.*

7.2 Castor Generalization

7.2.1 ARMG Algorithm

Castor modifies Algorithm 3 in Appendix A to compute equivalent $armg$s over composition/ decomposition. Before we explain the Castor generalization algorithm, we define some concepts. Given clause $\overrightarrow{C}$ and literal $R(u)$ in $\overrightarrow{C}$, we call u that may contain both variables and constants a *free tuple*. We extend the definitions of projection π and natural join $\bowtie$ operators over free tuples in natural manner. A *canonical database instance* of clause $\overrightarrow{C}$, shown as $I^{\overrightarrow{C}}$, is the database instance whose tuples are the free tuples in $\overrightarrow{C}$ [1]. In other words, relation I_R in $I^{\overrightarrow{C}}$ has free tuple u if literal $R(u)$ is in $\overrightarrow{C}$. In each iteration of the 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 3, 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 3 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 7.6. *Consider again the definitions for target relation $hardWorking$ from Example 6.4 over the Original and 4NF UW-CSE schemas in Table 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 satisfy $student(x)$ but does not satisfy $inPhase(x, prelim)$. Castor removes $inPhase$ 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. Hence, it returns equivalent generalizations.*

LEMMA 7.7. *The Castor ARMG is schema independent.*

7.2.2 Negative Reduction

Castor further generalizes clauses produced by ARMG by removing non-essential literals from clauses. 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 [18, 19]. This step is called *negative reduction* and reduces the generalization error of the produced definitions to the training data. Castor uses INDs with equality to compute equivalent reductions of clauses over composition/ 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 $\mathbf{N}$ 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 contain 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. Note that some literals in the discarded instances may also belong to other instances before or in Y_i. 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. More details can be found in Algorithm 4 in Appendix A.

LEMMA 7.8. *Algorithm 4 is schema independent.*

Based on Lemmas 7.5, 7.7, and 7.8, Castor is schema independent.

7.3 General Decomposition/ Composition

Castor is robust over schema variations caused by bijective decompositions and compositions as defined in Section 4. Bijective decompositions and compositions need at least one IND with equality in the transformed and original schemas, respectively. We have observed several examples of these transformations in real-world databases, some of which we report in Section 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 $S_1 \dots S_n$ that satisfies all conditions in Definition 4.1, but at least one IND in $\mathcal{S}$ (in the second condition of Definition 4.1) is an IND in 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 [1]. Consider the general decomposition from $\mathcal{R}_1$:$\{R_1(A, B, C)\}$ to $\mathcal{S}_1$:$\{S_1(A, B), S_2(A, C)\}$ with IND $S_2[A] \subseteq S_1[A]$, and the instance of $\mathcal{S}_1$ $I^1_{\mathcal{S}_1}$: $I^1_{S_1} = \{(a_1, b_1), (a_2, b_2)\}$, $I^1_{S_2} = \{(a_1, c_1)\}$. There is *not* any instance of $\mathcal{R}_1$ that represents the same information as $I^1_{\mathcal{S}_1}$. Hence, it is not clear how to define schema independence for $I^1_{\mathcal{S}_1}$. Also, the composition from $\mathcal{S}_1$ to $\mathcal{R}_1$ is not invertible as $I^1_{S_1} \bowtie I^1_{S_2}$ 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 [1, 7]. For example, one can compose instance $I^1_{\mathcal{S}_1}$ in the last example to $I^1_{\mathcal{R}_1}$: $\{(a_1, b1, c_1),\ (a_2, b2, x)\}$ where x is a labeled null that reflects the existence of an unknown value. However, it takes more than a single paper to define the semantic of learning over databases with labeled nulls and schema independence over transformations that introduce labeled nulls, so we leave this direction for future work. 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. Hence, the mapping between the instances in the original and the remaining instances of the transformed schemas is bijective, thus, it is definition bijective. We define hypothesis invariance and schema independence as defined in Section 3 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 its original schema so that it 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 $S_1 \dots S_n$ 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 $S_i[X] \subseteq S_j[X]$. Natural join between $S_1 \dots S_n$ does not lose any tuple in an instance of $\mathcal{S}$, $I_{\mathcal{S}}$, iff 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, it is definition bijective. Thus, hypothesis invariance and schema independence properties in Section 3 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/ composition. An algorithm is schema independent over a general decomposition/ composition if it is schema independent over its general decompositions and general compositions.

Consider again schema $\mathcal{S}$ with relation symbols $S_1 \dots S_n$. To achieve schema independence over general composition/ decomposition, given instance $I_{\mathcal{S}}$, 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 7.5, 7.7, and 7.8 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/ 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 preprocessing phase to finish. Another approach is to use INDs in 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 IND and write them by $\subseteq$ for brevity. 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 7.1. Since the natural join between relations in $\mathcal{S}$ is acyclic, the pairwise consistency implies the global consistency of the joint tuples. For the same reason, the proof of Proposition 7.4 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 the Castor 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, it is removed. 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 7.2.2 to use the new definition of inclusion class instance. This extension of Castor may not be schema independent as it may miss some tuples in bottom-up construction or ignore some literals in ARMG algorithms. For example, consider the general decomposition from $\mathcal{R}_1$:$\{R_1(A, B, C)\}$ to $\mathcal{S}_1$:$\{S_1(A, B), S_2(A, C)\}$ with IND $S_2[A] \subseteq S_1[A]$ and instances $J^1_{\mathcal{R}_1}$: $J^1_{R_1} = \{(a_1, b_1, c_1)\}$ and $J^1_{\mathcal{S}_1}$: $J^1_{S_1} = \{(a_1, b_1)\}$, $J^1_{S_2} = \{(a_1, c_1)\}$. Assume that the modified Castor bottom-clause construction over $J^1_{\mathcal{S}_1}$ 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 $J^1_{\mathcal{S}_1}$ and $J^1_{\mathcal{R}_1}$. Nonetheless, our empirical results in Section 8 show that this extension of Castor is more schema independent than other algorithms over general decomposition/ composition.

7.4 Castor Implementation

Current bottom-up algorithms do not run efficiently over medium or large databases because they must process long clauses [19]. 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 RDBMS VoltDB (*voltdb.com*). Relational databases are usually stored in RDBMS's. Therefore, it is natural to implement a learning algorithm on top of an RDBMS. Using an RDBMS also provides access to the schema constraints, e.g., inclusion dependencies, which we use to achieve schema independence. 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 updated database instance.

Coverage tests: Castor optimizes the generalization process by reducing the number of coverage tests. 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.

Efficiently evaluating clauses: 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 [28]. 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. Clause C *θ-subsumes* C' iff there is some substitution θ such that $C\theta \subseteq C'$. A ground bottom-clause is a bottom-clause that only contains constants. A candidate clause C covers example e iff C θ-subsumes the ground bottom-clause $\bot_e$ associated with e. Castor uses the efficient subsumption engine Resumer2 [16]. 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.

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. Castor minimizes clauses using theta-transformation [6]. 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.

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

8. EXPERIMENTS

8.1 Experimental Settings

We use three datasets whose statistics are shown in Table 2. The **HIV-Large** dataset contains information about 42,000 chemical compounds. We learn the target 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. Using these dependencies, we created two new schemas in 4NF, named 4NF-1 and 4NF-2. The Initial, 4NF-1 and 4NF-2 schemas are shown in Table 6 in Appendix B.2. 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 [23]. We learn the target relation *advisedBy(stud,prof)*, as explained in Section 1. The dataset comes with a set of constraints in form of first-order logic clauses that should hold over the dataset domain. Using these constraints, we iteratively compose the original schema to four different schemas, two of which are shown in Table 1.

The **IMDb** dataset contains information about movies. We learn the target relation *dramaDirector(director)*, which indicates that *director* has directed a drama movie. The original schema, called JMDB, is in 4NF. We transform the database to two new schemas called Denormalized and Stanford. The Stanford schema follows a structure similar to the one used in the Stanford Movie DB (*infolab.stanford. edu/pub/movies*). The three schemas are shown in Tables 9 and 10 in Appendix B.2. 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. More details about the datasets and transformations can be found in Appendix B.2.

We compare Castor to three relational learning systems: FOIL [21], Aleph [25], and GILPS [19]. **FOIL** system implements 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 ILP system that implements Progol by its default setting [17]. 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. Details of the parameter configuration are in Appendix B.1.

We compare the quality of the leaned definitions using *precision*

HIV-Large				
Algorithm	Metric	Initial	4NF-1	4NF-2
Aleph-FOIL	Precision	0.58	0.72	0
	Recall	0.42	0.91	0
	Time (hours)	3	0.9	6
Castor	Precision	0.81	0.81	0.81
	Recall	0.85	0.85	0.85
	Time (hours)	3.5	1.9	56
HIV-2K4K				
Aleph-FOIL	Precision	0.72	0.78	0
	Recall	0.69	0.81	0
	Time (min)	6.2	7.9	20.6
Aleph-Progol	Precision	0.70	0.79	-
	Recall	0.85	0.90	-
	Time (min)	58.5	72.2	> 75 h
Castor	Precision	0.80	0.80	0.80
	Recall	0.87	0.87	0.87
	Time (min)	15.1	6.5	335.5

Table 3: Results of learning relations over HIV-Large and HIV-2K4K data.

and *recall*. 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. We perform 5-fold cross validation for UW-CSE and 10-fold cross validation for HIV and IMDb datasets. We evaluate precision, recall, and running times, showing the average over the cross validation. All experiments were run on a server with 32 2.6GHz Intel Xeon E5-2640 processors, running CentOS Linux with 50GB of main memory.

8.2 Experimental Results

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. Different schemas cause Aleph-FOIL and Aleph-Progol to explore different regions of the hypothesis space. Aleph-FOIL cannot find any definition over the 4NF-2 schema of HIV-Large and HIV-2K4K datasets. 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 4, 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 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. This is reflected in their precision and recall, which are not significantly different across schemas. 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 5. 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.

Algorithm	Metric	Original	4NF	Denorm-1	Denorm-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 4: Results of learning relations over UW-CSE data.

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

Table 5: Results of learning relations over IMDb data.

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, it is hard to find a straightforward relationship between the style of design and effectiveness for these algorithms. For instance, Aleph-Progol delivers its highest precision over a denormalized schema, Denormalized-1, and its lowest precision for another denormalized schema, Denormalized-2, over UW-CSE. This relationship also varies based on the dataset. Aleph-FOIL delivers its highest precision over a normalized schema, original, over UW-CSE but its highest precision over IMDb is on a denormalized schema, i.e., Stanford. We observe a similar trend for recall.

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

General decomposition/ composition. To explore non-bijective decomposition/ compositions, we restore the INDs with equality that we have enforced on their schemas to their original forms. We run the extended version of Castor from Section 7.3 using the aforementioned INDs and all other regular INDs in each schema. The results are shown in Appendix B.3.

9. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under grant IIS-1423238 and by AFRL Contract No. FA8750-16-C-0044 (Raytheon BBN Technologies) under the DARPA BRASS program. We would like to thank Qiang Zeng and Jignesh Patel for providing us with the HIV database.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley, 1994.

[2] A. Abouzied, D. Angluin, C. Papadimitriou, J. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In *PODS*, 2013.

[3] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. Cafarella, A. Kumar, F. Niu, Y. Park, C. Re, and C. Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[4] M. Arias, R. Khardon, and J. Maloberti. Learning Horn expressions with LOGAN-H. *J. Mach. Learn. Res.*, 8:549–587, Dec. 2007.

[5] B. T. Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *TODS*, 38(4):28:1–28:31, 2013.

[6] V. S. Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. V. Laer. Query transformations for improving the efficiency of ILP systems. *J. Mach. Learn. Res.*, 4:465–491, Dec. 2003.

[7] R. Fagin. Inverting schema mappings. *TODS*, 32(4), 2007.

[8] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.

[9] W. Fan and P. Bohannon. Information Preserving XML Schema Embedding. *TODS*, 33(1), 2008.

[10] L. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. *VLDB Journal*, November 2015.

[11] L. Getoor and A. Machanavajjhala. Entity resolution in big data. In *KDD*, 2013.

[12] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12), 2012.

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

[14] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[15] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, 2015.

[16] O. Kuželka and F. Železný. A restarted strategy for efficient subsumption testing. *Fundam. Inf.*, 89(1):95–109, Jan. 2009.

[17] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13:245–286, 1995.

[18] S. Muggleton and C. Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.

[19] S. Muggleton, J. C. A. Santos, and A. Tamaddoni-Nezhad. Progolem: A system based on relative minimal generalisation. In *ILP*, volume 5989, 2009.

[20] J. Picado, A. Termehchy, and A. Fern. Schema independent relational learning. http://arxiv.org/abs/1508.03846, 2015.

[21] J. R. Quinlan. Learning Logical Definitions From Relations. *Machine Learning*, 5, 1990.

[22] L. D. Raedt. *Logical and Relational Learning*. Springer, 2008.

[23] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, Feb. 2006.

[24] O. Schulte. A tractable pseudo-likelihood function for Bayes nets applied to relational data. In *SIAM SDM*, pages 462–473, 2011.

[25] A. Srinivasan. *The Aleph Manual*, 2004.

[26] A. Termehchy, M. Winslett, and Y. Chodpathumwan. How Schema Independent Are Schema Free Query Interfaces? In *ICDE*, 2011.

[27] X. Yin, J. Han, J. Yang, and P. S. Yu. CrossMine: Effcient Classifcation Across Multiple Database Relations. In *ICDE*, 2004.

[28] Q. Zeng, J. M. Patel, and D. Page. Quickfoil: Scalable inductive logic programming. *PVLDB*, 2014.

APPENDIX

A. ALGORITHMS

Algorithm 3: ARMG algorithm.

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

Algorithm 4: Castor 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 $C'$
    $\mathbf{I} \leftarrow \mathbf{I}'$
```

B. EXPERIMENTAL DETAILS

B.1 Parameter Configuration

Machine learning algorithms usually require parameter tuning to run them successfully. We try to use the default parameter configuration for all systems, except when needed. 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 the number of negative examples covered by the clause. In FOIL, this value is set with the *aaccur* 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).

Top-down algorithms contain the parameter *clauselength*, which sets an upper bound on the number of literals in a clause. The default value for this parameter in Aleph is 4. Over HIV-Large and HIV-2K4K, the definition for the target relation must contain long clauses. With *clauselength* $= 4$, Aleph-FOIL and Aleph-Progol do not learn any clause. Therefore, we set this parameter to have values of 10 and 15.

B.2 Schemas

The HIV dataset was obtained from the National Cancer Institute's AIDS antiviral screen (*wiki.nci.nih.gov/display/NCIDTPdata*). The initial schema contains relation *compound(comp,atm)*, which indicates that compound *comp* contains atom *atm*. The schema 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)*. 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 INDs for the HIV-Large and HIV-2K4K datasets are shown in Tables 6 and 7, respectively. The HIV data contains 80 INDs in total.

The UW-CSE dataset was obtained from *alchemy.cs.washington.edu/data/uw-cse*. It 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 8 (top). The INDs in these constraints are *hasPosition[prof] = professor[prof], student[stud] = inPhase[stud], ta[crs] = taughtBy[crs], yearsInProgram[stud]* $\subseteq$ *student[stud]*, and *ta[stud]* $\subseteq$ *student[stud]*. According to the original set of constraints, if one considers only the professors whose position is *Faculty*, the IND *taughtBy[prof]* $=$ *professor[prof]* holds. If there are more INDs with equality in the schema, one can generate more schemas from the original UW-CSE schema using composition transformation. To evaluate the effectiveness of algorithms over more varieties of schemas, we have considered only professors with position *Faculty* to use the IND *taughtBy[prof] = professor[prof]*. For the same reasons, we also added the INDs *student[stud]* $\subseteq$ *yearsInProgram[stud]* and *courseLevel[crs]* $=$ *taughtBy[crs]* to the schema. We enforce the afore-mentioned constraints by removing a small fraction of tuples, 159 tuples, from the original dataset. All INDs for the UW-CSE dataset are shown in Table 8. Using these INDs, we iteratively compose the original schema to four different schemas, two of which are shown in Table 1. We compose *courseLevel* and *taughtBy* relations in 4NF schema to create the a more denormalized schema, named Denormalized-1, and compose *courseLevel*, *taughtBy*, and *professor* in 4NF schema to generate the fourth schema, named Denormalized-2.

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 form of *movies2X[Xid] =X[id]*, e.g.,
movies2director[directorid] = director[id]. To test over more transformations, we have changed some regular INDs in the database in form of *movies2X[id]* $\subseteq$ *movie[id]* to *movies2X[id] = movie[id]* where X is *genre*, *color*, *prodcompany*, *producer*, and *director* 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, e.g., composing *movies2director(id, directorid)* and *director(id, name)* into *movies2director(id, directorid, name)*, to create a new schema, called Denormalized. We use the second set of INDs with equality to compose 5 relations in JMDB schema, e.g., *movies2genre*, into *movie* relation 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*). We explored our JMDB database to find other INDs, which are listed in Table 11 in Appendix B. The three schemas and the full list of INDs in IMDb data are shown in Tables 9, 10 and 11.

Initial	4NF-1	4NF-2

bonds(bd,atm1,atm2)
bType1(bd,t1)
bType2(bd,t2)
bType3(bd,t3)
	bonds(bd,atm1,atm2,
	t1,t2,t3)
		bSource(bd,atm1)
		bTarget(bd,atm2)
		bType1(bd,t1)
		bType2(bd,t2)
		bType3(bd,t3)

	Common relations	
compound(comp, atm)	element_C(atm) ...	element_O(atm)
p2_0(atm)	p2_1(atm) ...	p3(atm)

Table 6: Schemas for the HIV-Large and HIV-2K4K datasets.

bonds[bd]=bType1[bd]
bonds[bd]=bType3[bd]
	bonds[bd]=bType2[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 7: The INDs in the initial HIV dataset.

B.3 Analysis

B.3.1 Performance Measures

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 or the lack of sufficient amount of training data. 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

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

JMDB	Stanford
movie(id,title,year)	movie(id,title,year,genreid, colorid,prodcompid, directorid,producerid)
movies2genre(id,genreid)	
movies2color(id,colorid)	
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 9: JMDB and Stanford schemas for the IMDb dataset. Relations in bottom are contained in both schemas.

be important in a financial application but vital in a medical application. Nevertheless, definitions with higher precision and/or recall are generally more desirable [21, 19, 25].

B.3.2 Robustness and Effectiveness

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.

Over the UW-CSE dataset, Aleph-FOIL learns definitions that overfit to the training data. These definitions vary with the different schemas. For instance, 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 unrestricted hypothesis space. Therefore, FOIL does not suffer from the same problems as Aleph-FOIL. However, it is less effective than other algorithms.

Denormalized	
movie(id,title,year)	language(id,language)
movies2actor(id,actorid,name,	plot(id,plot)
character,sex)	business(id,text)
movies2color(id,colorid,color)	altversion(id,version)
movies2X(id,Xid,name) s.t.	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)	

Table 10: Denormalized schema for the IMDb dataset.

movies2X[id] = movie[id]
s.t. X= {genre, color, prodcompany, producer, director}
movies2Y[Yid] = Y[id]
s.t. Y= {actor, cinematagr, color, composer, costdes, director, editor, misc, proddes, producer, writer}

Z[id] ⊆ movie[id] s.t. Z={business, runningtime, altversion, certificate, plot, rating, akatitle, distributor, releasedate, technical, movies2actor, movies2country, movies2composer, movies2writer, movies2costdes, movies2misc, movies2editor, movies2cinematgr, movies2language, movies2proddes} certificate[countryid] ⊆ country[countryid] releasedate[countryid] ⊆ country[countryid] akatitle[langid] ⊆ language[langid] movies2country[countryid] ⊆ country[countryid] movies2language[langid] ⊆ language[langid] movies2genre[genreid] ⊆ genre[genreid] movies2prodcompany[prdcompid] ⊆ prodcompany[prdcompid]

Table 11: The INDs in IMDB dataset

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.

B.3.3 Efficiency

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

HIV-Large				
Algorithm	Metric	Initial	4NF-1	4NF-2
Aleph-FOIL	Precision	0.68	0.68	0
$(clauselength = 15)$	Recall	0.41	0.85	0
	Time (hours)	11.7	3.7	47
HIV-2K4K				
Algorithm	Metric	Initial	4NF-1	4NF-2
Aleph-FOIL	Precision	0.70	0.78	0
$(clauselength = 15)$	Recall	0.79	0.89	0
	Time (min)	6.72	7.07	122.2
Aleph-Progol	Precision	0.72	0.75	-
$(clauselength = 15)$	Recall	0.89	0.87	-
	Time (min)	155.51	13.56	> 75 h

Table 12: Results of learning relations over HIV-Large and HIV-2K4K data with *clauselength* parameter set to 15.

becomes less efficient, as shown in Tables 12. Aleph-FOIL with both *clauselength* = 10 and 15 is also faster than Castor over the HIV-2K4K dataset, as shown in Tables 3 and 12. 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. This is exploited by related work that focuses on efficiency [28, 27, 10]. 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. This is reflected in our empirical studies, where 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 7.4. By reusing information about previous coverage tests, Castor reduces the number of coverage tests on new clauses. This 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.

The efficiency of Castor is comparable to the efficiency 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 impacted over the Denormalized-2 schema, as shown in Table 4. 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 5. 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.

HIV-2K4K				
Metric	Initial	4NF-1		4NF-2
Precision	0.77	0.79		0.73
Recall	0.63	0.87		0.76
Time (min)	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 (min)	7.3	90.8		8.1

Table 13: Results of Castor learning relations over HIV-2K4K, UW-CSE and IMDb data using only INDs in the form of subset.

B.3.4 General Decomposition/ Composition

To explore non-bijective decomposition/ compositions 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 8.1 for these datasets are general decomposition/ composition and not bijective. We run the extended version of Castor from Section 7.3 using the aforementioned INDs and all other regular INDs in each schema.

Table 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. For HIV-2K4K, it uses the INDs in Table 7 (bottom). For UW-CSE, it uses the INDs in Table 8 (bottom). For IMDb, it uses the INDs in Table 11 (bottom). The extension of Castor gets the same results as in Table 4 over UW-CSE and is schema independent. It is also robust and delivers the same results as in Table 5 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. This is because it cannot access some tuples in the bottom-clause construction in these databases as explained in Section 7.3. 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.