Noname manuscript No.
(will be inserted by the editor)

Logically Scalable and Efficient Relational Learning

Jose Picado · Arash Termehchy · Alan Fern · Parisa Ataei

Received: date / Accepted: date

Abstract Relational learning algorithms learn the definition of a new relation in terms of existing relations in the database. The same database may be represented under different schemas for various reasons, such as efficiency, data quality, and usability. Unfortunately, the output of current relational learning algorithms tends to vary quite substantially over the choice of schema, both in terms of learning accuracy and efficiency. We introduce 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 theoretically and empirically that current relational learning algorithms are generally not schema independent. We propose Castor, a relational learning algorithm that achieves schema independence.

1 Introduction

Learning novel concepts over relational databases has attracted a great deal of attention due to its applications in data management and machine learning [10, 28, 16]. Given a relational database and training examples for some target relation, *relational learning* algorithms attempt to find a definition of this relation in terms of the existing relations in the database [10, 25, 31]. Learned definitions are usually first-order logic formulas and often restricted to Datalog programs. For example, consider the UW-CSE database (*alchemy.cs.washington.edu/data/uw-cse*), which contains information about a computer science department and its schema fragments are shown in Table 1. Given the UW-CSE database and a set of student-advisor pairs, one may want to predict the new relation *advisedBy(stud,prof)*, which indicates that the student *stud* is advised by professor *prof*. A relational learning algorithm may learn the following Datalog program

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

which indicates that a student is advised by a professor if they have been co-authors of a publication.

One benefit of relational learning algorithms is that they can exploit the relational structure of the data. Moreover, their learned definitions are interpretable and easy to understand. Relational learning has several applications in database management and machine learning, such as learning database queries [2], learning new features [9], and learning the structure of statistical relational learning (SRL) models [15, 16].

Since the space of possible definitions (e.g. all Datalog programs) is enormous, relational learning algorithms must employ heuristics to search for effective definitions. Unfortunately, such heuristics typically depend on the precise choice of schema of the underlying database. This 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

J. Picado · A. Termehchy · A. Fern · P. Ataei
School of EECS, Oregon State University, Corvallis, OR, USA
E-mail: {picadolj,termehca,alan.fern,ataeip}@oregonstate.edu

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.

hasPosition. This would result in a more understandable schema with shorter query execution time, without introducing any redundancy.

Example 1 We use the classic relational learning algorithm FOIL [26] to learn a definition for the *advisedBy(stud, prof)* relation over the Original and 4NF schemas of the UW-CSE database, shown in Table 1. 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. On the other hand, FOIL learns the following Datalog rule over the 4NF schema:

$$advisedBy(x, y) \leftarrow student(x, post\_generals, 5), \\ professor(y, facul), publication(z, y), taughtBy(v, y, w),$$

which covers 12 positive examples and 10 negative examples. 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 [1, 13]. 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. 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 relations that a domain expert would judge as correct and relevant. 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 [6], and entity resolution [15]. 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 [13]. 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 tunning under a new schema, it may take a prohibitively long time to find the best schema. Further, since relational learning algorithms need to access the content of the database, one has to transform the underlying data to the desired schema, which may not be practical for a large and/or constantly evolving database.

In this paper, we introduce the 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 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 popular families of top-down [21,26] (Section 5) and bottom-up [22,23] learning algorithms (Section 6). We prove that they are *not* schema independent under (de) composition transformations.

- We introduce Castor, a bottom-up algorithm that is provably schema independent under (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 formalize the notion of schema independence for *query-based* learning algorithms, which learn the target concepts by asking queries to an *oracle*, e.g., a database user [18,2]. We prove that algorithms in this family are not schema independent (Section 8).
- 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 three real-world databases (Section 9). Our empirical results generally confirm our theoretical results and show that Castor is more efficient and as effective as, or more effective than, current algorithms.

2 Background

2.1 Related Work

There has been a growing interest in developing relational learning algorithms that scale to large databases [14,7,31]. AMIE [14] and Ontological Pathfinding [7] focus on learning first-order rules from RDF-style knowledge bases. They impose several restrictions to the learned rules to be able to learn over large knowledge bases. QuickFOIL [31] provides an in-RDBMS implementation of a modified version of FOIL. Besides efficiency, we also focus on schema independence.

Davis et al. [9] use a relational learning algorithm to learn new features, which are used to improve the performance of the SRL model. The proposed system in this paper can also be used to learn these features as well as the structure of the SRL models, with the added benefit of efficiency and schema independence.

We build upon the body of work on transforming databases without modifying their content by exploring the sensitivity of relational learning algorithms to such transformations [17,13]. Another notable group of database transformations is schema mapping for data exchange [12]. But, these transformations generally lose information and introduce incomplete information to a database. Researchers have defined the property of design independence for keyword query processing over XML [30]. We extend this line of work by exploring the schema independence for relational learning algorithms.

This paper extends previous work [25] in several directions. First, it contains the proofs for some of our theoretical results. The proofs for other results are in [24]. Second, it analyzes the schema independence of Golem [22], a well-known bottom-up relational learning system. Third, it provides the theoretical and empirical analyses of query-based relational learning algorithms. Fourth, it explains an extension of Castor [25] that forces learned definitions to be safe, i.e., all head variables must appear in the body. This allows Castor to learn from positive examples only. Finally, it empirically evaluates the impact of Castor's design choices, namely parallelization and stored procedures.

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 relation symbols and *constraints*, respectively. A constraint restricts the properties of data stored in a database. Examples of constraints are *functional dependencies* (FD) and *inclusion dependencies* (IND), i.e., referential integrity. 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_i's are constants, the atom is a *ground atom*. A *literal* is an atom, or the negation of an atom. A *ground literal* is a literal whose atom is a ground atom. A *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 bodies 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. The heads of all clauses in these definitions are the *target relation*.

3 Framework

3.1 Relational Learning

Relational learning algorithms learn first-order definitions from training examples and a relational database. Training examples E are usually tuples of a single target relation T, which express positive (E^+) or negative (E^-) examples. The input database instance I is also called *background knowledge*. The learned definition is called the hypothesis H, which is usually restricted to Horn definitions for efficiency reasons. In the following sections we provide concrete definitions of several relational learning algorithms.

Example 2 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

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

which 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. We denote the hypothesis space (or language) of algorithm A over schema $\mathcal{R}$ with parameters θ as $\mathcal{L}^A_{\mathcal{R},\theta}$. The hypothesis space $\mathcal{L}^A_{\mathcal{R},\theta}$ is a subset of $\mathcal{HD}_{\mathcal{R}}$ [21,26].

Example 3 Continuing Example 2, consider restricting the hypothesis space to clauses whose number of literals are fewer than a given number, which we call clause-length. Assume that we are now interested in learning a definition for the target relation *collaboratedProf(x,y)*, which indicates that professor x has collaborated with professor y, under the Original schema. If we set clause-length = 5, the learning algorithm is able to learn the definition

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

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

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 [17,13].

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 *total* iff it is defined for every element of $\mathcal{I}(\mathcal{R})$. 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 [17,13]. 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 4 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 1 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 [13] 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 5 Let $\mathcal{R}$ be the Original schema and $\mathcal{S}$ be the 4NF schema in Example 4. The transformation from the Original schema to the 4NF schema can be written as the following set of Horn definitions:

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

The inverse of this transformation from the 4NF to Original schema is a set of projection operators, which can also be written as 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 1 *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 transformations, every Horn definition in $\mathcal{HD}_\mathcal{R}$ can be rewritten as a Horn definition in $\mathcal{HD}_\mathcal{S}$ such that they return the same results over equivalent database instances. Hence, in the rest of this paper, we consider only the bijective Horn transformations whose inverses are Horn transformations.

Example 6 Let $\mathcal{R}$ be the Original schema and $\mathcal{S}$ be the 4NF schema in Example 4, and $\tau : \mathcal{R} \to \mathcal{S}$ and $\tau^{-1} : \mathcal{S} \to \mathcal{R}$ are the Horn transformation explained in Example 5. According to Proposition 1, 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. Example 3 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 2 Algorithm A is hypothesis invariant under transformation $\tau : \mathcal{R} \to \mathcal{S}$ 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. A relational learning algorithm $A(I, E, \theta)$ takes as input a database instance I, training examples E, and parameters $\theta \in \Theta$, and outputs a hypothesis in $\mathcal{L}^A_{\mathcal{R},\theta}$.

Definition 3 Algorithm A is schema independent under bijective transformation $\tau : \mathcal{R} \to \mathcal{S}$ 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 7 Consider the Original schema and the 4NF schema in in Example 4. 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, 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, and is not schema independent.

4 Decomposition and Composition

There are many bijective Horn transformations between relational schemas [17,1]. 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 de-normalizations, 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 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

1. For each relation I_R there is one and only one instance $(I_{S_1} \ldots I_{S_n})$ of $\mathcal{S}$ such that $\pi_{sort(S_i)}(I_R) = I_{S_i}$, $1 \leq i \leq n$, and $\bowtie_{i=1}^{n} I_{S_i} = I_R$.
2. For all S_i, S_j, $1 \leq i, j \leq n$, such that $X = sort(S_i) \cap sort(S_j) \neq \emptyset$, $\Sigma_{\mathcal{S}}$ contains IND with equality $S_i[X] = S_j[X]$.
3. We have $\Sigma_S = \Sigma_R \cup \lambda$, where λ is the set of INDs with equality in the second condition.

The first and third conditions in Definition 4 are generally known as *lossless join* and *dependency preservation* properties, respectively. The second condition in Definition 4 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, 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. 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 relations has a dangling tuple regarding the join. For example, the join between the relations of $\mathcal{S}$ in the first condition of Definition 4 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 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

Many relational learning algorithms follow a covering approach [26,21]. 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. In top-down algorithms, the *LearnClause* function searches the hypothesis space from general to specific. The hypothesis space in top-down algorithms is a tree in which nodes represent clauses and each edge is the application of a basic refinement operator, which generally consists of adding a new literal to the clause. Top-down algorithms start from the most general clause, which corresponds to the root of the tree, and repeatedly refine it until reaching some stopping condition.

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 strategy of searching the tree varies between different top-down algorithms. For instance, FOIL [26, 31] is an efficient and popular top-down algorithm that follows a greedy best-first search strategy. Given the current clause, FOIL specializes a clause by adding the literal that provides the most information gain. FOIL stops adding literals to the clause when the number of bits required to encode the clause exceeds the number of bits required to indicate the number of positive examples covered by the clause. Progol [21] is another well-known top-down algorithm similar to FOIL, except that it does not follow a greedy search strategy, and it restricts the literals that can be added to the clause. Further, Progol limits the length of the clause, i.e., the maximum number of literals in a clause.

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

Theorem 1 *FOIL is not hypothesis invariant.*

Proof Let $\mathcal{R}$ be a schema, E be the training data, C be a clause, n be the number of variables in C, and p be the number of positive examples covered by C. The number of bits required to indicate that these examples are covered by C is $b_e(C) = \log_2(|E|) + \log_2(\binom{|E|}{p})$ [26].

The number of bits $b_c(C)$ required to encode clause C is equal to the sum of the number of bits required to encode each literal in C, reduced by $\log_2(m!)$, where m is the number of literals in C. The number of bits required to encode a literal is $1+\log_2(|\mathcal{R}|)+\log_2(n)$ [26]. A clause C is in hypothesis space $\mathcal{L}$ if $b_c(C) \leq b_e(C)$.

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

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

One may want to fix the problem of schema dependence in Progol by choosing different values for the maximum lengths over the original and transformed schemas. The following theorem 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. The proof can be found in [24].

Theorem 2 *Progol is not hypothesis invariant.*

6 Bottom-up Algorithms

Bottom-up algorithms also follow the covering approach shown in Algorithm 1. However, their *LearnClause* function searches the hypothesis space from specific to general hypotheses. Given a positive example, bottom-up algorithms attempt to find the most specific clause in the hypothesis space, called *bottom-clause*, that covers the example, relative to the database instance [22,23]. Then, they generalize these bottom-clauses to find definitions that cover as most positive and as fewest negative examples as possible. There are multiple bottom-up algorithms whose differences lie mainly in their generalization operator [22,23,4]. We consider two algorithms that are representative of the family of bottom-up algorithms: Golem [22] and ProGolem [23].

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 [21]. 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 tuple.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 [21]. 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 8 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 9 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 8 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 8 is not in the hypothesis language. But, under the new schema, the clause presented above is in the hypothesis language.

The following lemma proves that the bottom-clause construction algorithm is schema dependent even if different depth values are used across schemas.

Lemma 1 *Bottom-clause construction is not schema independent.*

6.2 Golem

In this section, we consider a bottom-up learning algorithm called Golem [22]. Golem, like other learning algorithms, follows a covering approach, as the one shown in Algorithm 1. Golem's *LearnClause* function follows a bottom-up approach, which is based on the *least general generalization* (lgg) operator. Given clauses C_1 and C_2, the lgg of C_1 and C_2 is the clause C that is more general than C_1 and C_2, but the least general such clause. The notion of generality is defined by θ-subsumption. Therefore, clause C is more general than C_1 if and only if C θ-subsumes C_1 (and similarly for C_2). This notion of generality gives a computable generality relation. Further, the lgg of two clauses is unique. Because of the lack of space, for further details we refer the readers to [22].

Golem uses a special case of bottom-clause, where all literals of the clause are grounded. We call this type of clause the *saturation*. A saturation can be computed using the bottom-clause construction algorithm described above. Given the saturations for a pair examples, the operator that computes the lgg for the pair of saturations is called the *relative least general generalization* ($rlgg$). The lgg of a set of saturations is defined via pairwise operations, that is

$$lgg(\{C_1, ..., C_n\}) = lgg(lgg(\{C_1, ..., C_{n-1}\}), C_n)$$

The order of pairwise lggs does not matter as the lgg operator is commutative and associative.

Given a database instance I and training examples E^+ and E^-, Golem's *LearnClause* function learns a clause that covers the most positive and the fewest negative examples as possible. Algorithm 2 sketches this function. Intuitively, the algorithm first randomly selects a subset E_S^+ of positive examples E^+. It then generates candidate clauses by computing the $rlgg$ between every pair of examples in E_S^+. The algorithm considers only candidate clauses that satisfy some minimum condition, e.g., minimum precision of a clause. It then greedily includes new examples into the generalization to create new candidate clauses. This algorithm uses the function $Covers(C, E)$, which returns the examples in E covered by clause C. The algorithm stops when no improvement can be done.

Algorithm 2: Golem's *LearnClause* algorithm.

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

Theorem 3 *The rlgg operator is schema independent.*

Proof Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a bijective transformation that is a vertical composition/ decomposition between schemas $\mathcal{R} = (\mathbf{R}, \Sigma_\mathcal{R})$ and $\mathcal{S} = (\mathbf{S}, \Sigma_\mathcal{S})$. Let I and J be instances of $\mathcal{R}$ and $\mathcal{S}$, respectively, such that $\tau(I) = J$. Let T be the target relation, and $e_1 = T(a_1, \cdots, a_l)$ and $e_2 = T(b_1, \cdots, b_l)$ be two positive examples. Let $(e_1 \leftarrow I_1')$ and $(e_2 \leftarrow I_2')$ be the saturations under schema $\mathcal{R}$ for e_1 and e_2, respectively, such that $I_1', I_2' \subseteq I$. Similarly, let $(e_1 \leftarrow J_1')$ and $(e_2 \leftarrow J_2')$ be the saturations under schema $\mathcal{S}$ for e_1 and e_2, respectively, such that $J_1', J_2' \subseteq J$.

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

$$rlgg_\mathcal{R}(e_1, e_2) \equiv rlgg_\mathcal{S}(e_1, e_2)$$

$$lgg((e_1 \leftarrow I_1'), (e_2 \leftarrow I_2')) \equiv lgg((e_1 \leftarrow J_1'), (e_2 \leftarrow J_2'))$$

We know that $(e_1 \leftarrow I_1')$ and $(e_2 \leftarrow I_2'))$ are clauses. Therefore, $lgg((e_1 \leftarrow I_1'), (e_2 \leftarrow I_2'))$ is the set of pairwise lgg operations of compatible ground atoms in $(e_1 \leftarrow I_1')$ and $(e_2 \leftarrow I_2')$. Two atoms are compatible if they have the same relation name. We show that the lgg of compatible ground atoms under schema $\mathcal{R}$ delivers equivalent results under schema $\mathcal{S}$.

Let $R \in \mathbf{R}$ be a relation in $\mathcal{R}$ such that $\tau(R) = S_1, \cdots, S_m$, $1 \leq m \leq |\mathbf{S}|$. Because of Corollary 4.3.2 in [5], we know that if τ is bijective, $\Sigma_\mathcal{S}$ contains inclusion dependencies between the join attributes of $S_1, \cdots, S_m$. Let $r_1 = R(a_1, \cdots, a_k)$ and $r_2 = R(a_1', \cdots, a_k')$ be two ground atoms in I. Then, $\tau(r_1) = S_1(t_1), \cdots, S_m(t_m)$ and $\tau(r_2) = S_1(t_1'), \cdots, S_m(t_m')$ are ground atoms in J,

where t_i and t'_i, $1 \leq i \leq m$, are tuples. Then, the lgg of ground atoms r_1 and r_2 is defined as

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

By applying transformation τ, this is equivalent to

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

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

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

□

In Section 7.1 we show that the bottom-clause construction algorithm can be modified to be schema independent. Because the $rlgg$ operator is also schema independent, Golem can achieve schema independence. However, Golem may generate very large clauses after each application of the $rlgg$ operator. The reason is that the size of a clause generated by $lgg(C_1, C_2)$, where C_1 and C_2 are clauses, is bounded by $|C_1| \cdot |C_2|$. Let n be the number of positive examples to generalize and m be the maximum length of a bottom-clause. Then, the length of the clause generated by $rlgg$ is bounded by $O(m^n)$, i.e., it grows exponentially in the number of positive examples covered. This results in exponential running time. Therefore, an algorithm that uses the $rlgg$ operator, such as Golem [22], cannot learn efficiently without making assumptions that do not hold over most real-world databases [23].

6.3 ProGolem

ProGolem is a bottom-up algorithm that can run efficiently over small or medium databases without making generally unrealistic assumptions [23]. To explore the hypothesis space and generalize clauses efficiently, ProGolem assumes that clauses are ordered. An *ordered clause* is a clause where the order and duplication of literals matter. If clause C is considered an ordered clause, then it is denoted as $\overrightarrow{C}$. For instance, clauses $\overrightarrow{C} = T(x) \leftarrow P(x), Q(x)$, $\overrightarrow{D} = T(x) \leftarrow Q(x), P(x)$, and $\overrightarrow{E} = T(x) \leftarrow P(x), P(x), Q(x)$ are all different.

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

Algorithm 3: *ARMG* algorithm.

Input : Bottom-clause $\bot_{e,I_\mathcal{R}}$, positive example e'.
Output: An *ARMG* of $\bot_{e,I_\mathcal{R}}$ that covers e'.
$\overrightarrow{C}$ is $\bot_{e,I_\mathcal{R}} = T \leftarrow L_1, \cdots, L_n$
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 $\overrightarrow{C}$

We now explain the *armg* operator in detail. Let $\bot_{e,I_\mathcal{R}}$ be the bottom-clause associated with example e, relative to $I_\mathcal{R}$. Let $\overrightarrow{C} = T \leftarrow L_1, \cdots, L_n$ be the ordered version of $\bot_{e,I_\mathcal{R}}$. 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 $I_\mathcal{R}$ [23]. Algorithm 3 shows the *ARMG* algorithm, which implements the *armg* operator. Given the bottom-clause $\bot_{e,I_\mathcal{R}}$ and a positive example e', *armg* drops all blocking atoms from the body of $\bot_{e,I_\mathcal{R}}$ until e' is covered. After removing a blocking atom, some literals in the body may not have any variable in common with the other literals in the body and head of the clause, i.e., they are not *head-connected*. *ARMG* also drops 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 10 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 4 *ProGolem is not schema independent.*

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, they can be used by the algorithm for improving the effectiveness of the learned definitions.

Castor's *LearnClause* function is shown in Algorithm 4. 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 7.5. 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 4: Castor's *LearnClause* algorithm.

```
Input : Database instance I, positive examples E+,
        negative examples E−, parameters K and N.
Output: A new clause C.
C⃗ ← Castor_BottomClause(first example in E+)
C⃗ ← Minimize(C⃗) ; BC ← {C⃗}
repeat
    BestScore ← highest score of candidates in BC
    E+_S ← K randomly selected positive examples
      from E+
    NC = {}
    foreach clause C ∈ BC do
        foreach e′ ∈ E+_S do
            C′ ← Castor_ARMG(C, e′)
            if Score(C′) > BestScore then
                NC ← NC ∪ C′
    BC ← highest scoring N candidates from NC
until NC = {}
C′ ← 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 $S_1 \dots S_n$ 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 $s_1, \dots, s_n$ in instances $I_{S_1}, \dots, I_{S_n}$, 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 $S_1 \dots S_n$ 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, there are INDs with equality between the join attributes of relation symbols $S_1 \dots S_n$. We use INDs with equality between the attributes in schema $\mathcal{S}$ to find tuples s_i. To simplify our notations, we assume that the join between relations in $\mathcal{S}$ is still natural join. Our results extend for composition joins that are equi-join.

Definition 5 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 all tuples $s_1, \ldots, s_n$ that participate in the join and none of them is a dangling tuple with regard to the full join. Otherwise, Castor must check the join condition for each pair of tuples.

Example 11 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 :*yearsInProgram(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 6 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 $S_1[B] = S_2[B]$, $S_2[C] = S_3[C]$, and $S_3[A] = S_1[A]$ is cyclic. Consider tuples s_1, s_2, and s_3 such that $\pi_B(s_1) = \pi_B(s_2)$ and $\pi_C(s_2) = \pi_C(s_3)$. We may not have $\pi_A(s_3) = \pi_A(s_1)$. Hence, Castor has to scan many tuples in S_3 to find a tuple s'_3 that satisfies both $\pi_C(s_2) = \pi_C(s'_3)$ and $\pi_A(s'_3) = \pi_A(s_1)$. The following proposition shows that if the composition join in Definition 4 is acyclic, the INDs with equality in the decomposed schema are not cyclic. Thus, Castor does not face the aforementioned issue.

Proposition 2 *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 are not cyclic.*

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

Given $S_i, S_j \in \mathbf{N}$, too many tuples from a relation I_{S_j} may join with the current tuple $s_i \in I_{S_i}$, which may result in an extremely large bottom-clause. One may limit the maximum number of tuples that can join with the current tuple to a reasonably large value. We use the value of 10 in our reported experiments. After finding the 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 7.5 and 9 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 2 *Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a composition/ decomposition, I be an instance of $\mathcal{R}$, and $\perp_{e,I}$ and $\perp_{e,\tau(I)}$ are*

bottom-clauses generated by Castor for example e relative to I and $\tau(I)$, respectively. We have $\bot_{e,I} \equiv \bot_{e,\tau(I)}$.

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

7.2 Castor Generalization

7.2.1 ARMG Algorithm

Castor modifies Algorithm 3 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 12 Consider again the definitions for target relation *hardWorking* from Example 10 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 satisfies $student(x)$ 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 3 *Castor's ARMG is schema independent.*

Proof Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a decomposition from schema $\mathcal{R} = (\mathbf{R}, \Sigma_{\mathcal{R}})$ and $\mathcal{S} = (\mathbf{S}, \Sigma_{\mathcal{S}})$. Let τ map each relation $R_i \in \mathbf{R}$ to relations $S_{i_1} \ldots S_{i_m} \in \mathbf{S}$. Assume that the input to the *ARMG* algorithm over schema $\mathcal{R}$ is the bottom-clause for seed example e, denoted as $\overrightarrow{C_{\mathcal{R}}}$, which is in the form of $T(w) \leftarrow L_1(u_1), \cdots, L_n(u_n)$. The input to the algorithm over schema $\mathcal{S}$ is the bottom-clause for seed example e, denoted as $\overrightarrow{C_{\mathcal{S}}}$, which is in the following form: $T(w) \leftarrow P_1(v_1), \cdots, P_k(v_k)$. $\overrightarrow{C_{\mathcal{R}}}$ and $\overrightarrow{C_{\mathcal{S}}}$ are generated by the Castor bottom-clause construction algorithm and according to Lemma 2 are equivalent. They also do not contain any redundant literal.

The mapping between equivalent clauses over $\mathcal{R}$ and $\mathcal{S}$, δ_τ, that is associated with τ projects each literal with relation symbol R_i in $\overrightarrow{C_{\mathcal{R}}}$ to literals with relation symbols $S_{i_1} \ldots S_{i_m}$ in the clause $\overrightarrow{C_{\mathcal{S}}}$. Hence, there is a bijective mapping M that maps each literal $R_i(u_l)$ in the body of $\overrightarrow{C_{\mathcal{R}}}$ to a set of literals $S_{i_1}(v_j) \ldots S_{i_m}(v_{j+(i_m - i_1)})$ in the body of $\overrightarrow{C_{\mathcal{S}}}$. Moreover, according to Lemma 2, a literal L_l appears before L_o in the body of $\overrightarrow{C_{\mathcal{R}}}$ iff all literals in $M(L_l)$ appear before the ones in $M(L_o)$ in $\overrightarrow{C_{\mathcal{S}}}$. The mapping δ only projects each literal with relation symbol $R_i(u_l)$ to a set of literals in $M(R_i(u_l))$. Hence, the free tuples in every pair of literals L_l and L_o in $\overrightarrow{C_{\mathcal{R}}}$ have a variable in common iff the sets of free tuples in $M(L_l)$ and $M(L_o)$ have a shared variable. Otherwise, $\overrightarrow{C_{\mathcal{R}}}$ and $\overrightarrow{C_{\mathcal{S}}}$ are not equivalent.

Assume that Castor removes literal L_b in $\overrightarrow{C}_{\mathcal{R}}$ because it is the blocking atom in the current iteration. Let the positive example considered for this iteration of the algorithm be e'. If L_b is the blocking atom, the sub-clause of $\overrightarrow{C}_{\mathcal{R}}$ up to and excluding L_b covers e' and

the one up to and including L_b does not cover e'. Because mapping M preserves the order of literals, the sub-clause of $\overrightarrow{C}_{\mathcal{S}}$ up to and excluding L_b covers e' and the one up to and including literals in $M(L_b)$ does not cover it. Hence, at least one literal in $M(L_b)$ is a blocking atom in $\overrightarrow{C}_{\mathcal{S}}$. If the algorithm removes this literal, it also drops the rest of literals in $M(L_b)$. This is because the free tuples of these literals do not satisfy the IND between relation symbols of $M(L_b)$ in the canonical database instance of $\overrightarrow{C}_{\mathcal{S}}$ after removing the blocking atom in $\overrightarrow{C}_{\mathcal{S}}$. Similarly, if one of the literals in $M(L_b)$ is a blocking atom, L_b will be also a blocking atom. In this case, the *ARMG* algorithm will also remove the non-blocking atoms in $M(L_b)$ that are not member of $M(L_o)$, $L_b \neq L_o$ as they do not satisfy any IND after removing the blocking atom.

Assume that a literal L_l is removed because it does not satisfy any IND in the canonical database instance of $\overrightarrow{C}_{\mathcal{R}}$ immediately after dropping the blocking atom L_b. Let the IND between the relation symbol of L_b and the relation symbol of L_l be Σ_1. Because τ preserves the INDs between relations in $\mathbf{R}$, there is also an IND Γ_1 between the relation symbol of a literal P_l in $M(L_l)$ and the relation symbol of a literal in $M(L_b)$. Because L_b is a blocking atom, *ARMG* algorithm has already removed all literals in $M(L_b)$ from $\overrightarrow{C}_{\mathcal{S}}$. Assume that the free tuples of P_l and another literal P_o in $\overrightarrow{C}_{\mathcal{S}}$ satisfy Γ_1. If P_o has not been already removed from $\overrightarrow{C}_{\mathcal{S}}$, the free tuples of L_l and L_o satisfy the IND constraint Σ_1 in the canonical database of $\overrightarrow{C}_{\mathcal{R}}$. Thus, L_l should not have been removed from $\overrightarrow{C}_{\mathcal{R}}$. Therefore, P_o is removed from $\overrightarrow{C}_{\mathcal{S}}$. Hence, P_l must also be removed from $\overrightarrow{C}_{\mathcal{S}}$ as it does not satisfy any IND. After removing P_1, all literals in $M(L_l)$ will be removed from $\overrightarrow{C}_{\mathcal{S}}$. Using similar argument, we show that if the *ARMG* algorithm removes a literal L_r from $\overrightarrow{C}_{\mathcal{R}}$ because its free tuple does not satisfy any IND after dropping another literal, the algorithm removes the literals in $M(L_r)$ that are not member of $M(L_o)$, $L_r \neq L_o$. Also, we prove that if the algorithm eliminates a literal P_r from $\overrightarrow{C}_{\mathcal{S}}$ because its free tuple does not satisfy any IND, the algorithm also removes the literals L_r, where $P_r \in M(L_r)$ from $\overrightarrow{C}_{\mathcal{R}}$. We similarly prove that if Castor removes a literal because it is not head-connected, it also removes its corresponding literals over the decomposition and vice versa. □

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 [22,23]. 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

Algorithm 5: 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'
  $\mathbf{I} \leftarrow \mathbf{I'}$

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 5.

Lemma 4 *Castor's negative reduction is schema independent.*

Proof Let $\tau : \mathcal{R} \rightarrow \mathcal{S}$ be a composition/ decomposition between schemas $\mathcal{R} = (\mathbf{R}, \Sigma_\mathcal{R})$ and $\mathcal{S} = (\mathbf{S}, \Sigma_\mathcal{S})$. Let $R[U] \in \mathbf{R}$ and τ map relation $R[U]$ to relations $S_1[V_1], \cdots, S_m[V_m]$, $1 \leq m \leq |\mathbf{S}|$. Let $\mathbf{N}$ be the inclusion class in $\Sigma_\mathcal{S}$ that contains relations $S_1[V_1], \cdots, S_m[V_m]$. Assume that $\overrightarrow{C}_\mathcal{R}$ is a clause over schema $\mathcal{R}$ and contains k literals $R(u_i)$, $1 \leq i \leq k$. Let $\overrightarrow{C}_\mathcal{S}$ be the equivalent clause of $\overrightarrow{C}_\mathcal{R}$ over $\mathcal{S}$. Let *Reduce*(C) be the function that performs negative reduction on clause C. We show that *Reduce*$(\overrightarrow{C}_\mathcal{R}) \equiv$ *Reduce*$(\overrightarrow{C}_\mathcal{S})$.

Because $\overrightarrow{C}_\mathcal{R}$ contains k literals $R(u_i)$, $1 \leq i \leq k$, and $\overrightarrow{C}_\mathcal{R} \equiv \overrightarrow{C}_\mathcal{S}$, then $\overrightarrow{C}_\mathcal{S}$ must contain k instances of inclusion class $\mathbf{N}$. These instances of inclusion class may or may not share literals. Let n be the number of instances of inclusion class $\mathbf{N}$ in $\overrightarrow{C}_\mathcal{S}$ that share literals. Without loss of generality, we assume that instances can only share the first literal. That is, instances $I_{\mathbf{N}i}$ and $I_{\mathbf{N}j}$ share a literal if they have the form $I_{\mathbf{N}i} = S_1(v_{\_1}), S_2(v_{i2}), \cdots, S_m(v_{im})$ and $I_{\mathbf{N}j} = S_1(v_{\_1}), S_2(v_{j2}), \cdots, S_m(v_{jm})$. We prove by induction on n.

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

Assumption step: let $n = k$. $\overrightarrow{C}_\mathcal{R}$ contains literals $[R(u_i)]$, $1 \leq i \leq k$, $\overrightarrow{C}_\mathcal{S}$ contains literals $S_1(v_{\_1})$, $[S_2(v_{i2}), \cdots, S_m(v_{im})]$, $1 \leq i \leq k$ and $\overrightarrow{C}_\mathcal{R} \equiv \overrightarrow{C}_\mathcal{S}$.

Induction step: let $n = k+1$. Let $\overrightarrow{C}_\mathcal{S}$ contain $k+1$ instances of inclusion class $\mathbf{N}$, which share the first literal. Let $\overrightarrow{C}_\mathcal{R}$ be the equivalent clause, which contains $k+1$ literals $R(u_i)$, $1 \leq i \leq k+1$. We divide instances in $\overrightarrow{C}_\mathcal{S}$ in two: $\mathbf{I}_{\mathbf{N}(\mathbf{1..k})} = S_1(v_{\_1}), [S_2(v_{i2}), \cdots, S_m(v_{im})]$, $1 \leq i \leq k$ and $I_{\mathbf{N}(k+1)} = S_1(v_{\_1}), S_2(v_{(k+1)2}), \cdots, S_m(v_{(k+1)m})$. We also divide literals in $\overrightarrow{C}_\mathcal{R}$ in two: $\mathbf{R}_{\mathbf{1..k}} = [R(u_i)]$, $1 \leq i \leq k$ and $R(u_{k+1})$.

Let $\overrightarrow{C'}_\mathcal{S}$ contain all literals in $\mathbf{I}_{\mathbf{N}(\mathbf{1..k})}$ and $\overrightarrow{C'}_\mathcal{R}$ contain all literals in $\mathbf{R}_{\mathbf{1..k}}$. We examine the cases where we add literal $R(u_{k+1})$ to $\overrightarrow{C'}_\mathcal{R}$ such that $\overrightarrow{C'}_\mathcal{R} \cup \{R(u_{k+1})\} = \overrightarrow{C}_\mathcal{R}$, and we add all literals in instance $I_{\mathbf{N}(k+1)}$ to $\overrightarrow{C'}_\mathcal{S}$ such that $\overrightarrow{C'}_\mathcal{S} \cup I_{\mathbf{N}(k+1)} = \overrightarrow{C}_\mathcal{S}$.

Castor removes all literals in $\mathbf{R}_{\mathbf{1..k}}$ and literal $R(u_{k+1})$ iff it removes all literals in $\mathbf{I}_{\mathbf{N}(\mathbf{1..k})}$ and $I_{\mathbf{N}(k+1)}$. Then, *Reduce*$(\overrightarrow{C}_\mathcal{R}) \equiv$ *Reduce*$(\overrightarrow{C}_\mathcal{S})$.

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

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

Based on Lemmas 2, 3, and 4, Castor is schema independent.

7.3 Generating Safe Clauses

Let the head-variables of a clause be the ones that appear in its head literal. A clause is *safe* if every head-variable appears in some literal in the body of the clause. A definition is safe if all its clauses are safe. The results of safe clauses and definitions are finite over a (finite) database. By default, current relational learning algorithms, including Castor, may learn *unsafe* Datalog definitions [1]. Because an unsafe definition produces infinitely many answers over a (finite) database, it is *not* desirable in many relevant applications, such as learning database queries from examples [20,2]. Furthermore, a relational learning algorithm that learns only safe clauses can learn a definition from positive examples only. In this section, we describe how Castor can be modified to generate only safe definitions. As we have explained, Castor first constructs the bottom-clause associated with some positive example e, and then generalizes this clause using *ARMG* and negative reduction.

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

Safe ARMG Algorithm: Let the *ARMG* algorithm take as input clause $\overrightarrow{C}$ and positive example e,

and produce as output clause $\overrightarrow{C'}$. Clause $\overrightarrow{C'}$ may not be safe. Castor checks whether $\overrightarrow{C'}$ is safe. If $\overrightarrow{C'}$ is safe, Castor considers it as a candidate; otherwise, Castor simply ignores it.

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

7.4 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 9. 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 but at least one IND in $\mathcal{S}$ (in the second condition of Definition 4) 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 [1,11]. 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 $S_1 \ldots S_n$. To achieve schema independence over general composition/ decomposition, given instance $I_{\mathcal{S}}$, Castor finds each INDs $S_i[X] \subseteq S_j[X]$ 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 2, 3, and 4 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 pre-processing 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 2 extends for INDs. Hence, the INDs in each inclusion class are not cyclic and Castor efficiently finds the tuples that join according to the INDs. We also extend Castor's *ARMG* algorithm to ensure that the free tuple of each literal $S(u)$, u, satisfies all INDs in which S participates after a blocking atom is removed. If u does not satisfy any of its corresponding INDs, 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 $J^1_{\mathcal{R}_1}$. But, our empirical results in Section 9 show that this extension of Castor is more schema independent than other algorithms over general decomposition/ composition.

7.5 Castor System Design Choices and Implementation

Current bottom-up algorithms do not run efficiently over medium or large databases because they produce many long bottom-clauses to generalize [23]. Also, these clauses are time-consuming to evaluate. A relational learning algorithm evaluates a clause by computing the number of positive and negative examples covered by the clause. These tests dominate the time for learning [10]. 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 RDBMSs. Therefore, it is natural to implement a relational 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. Castor performs bottom-clause construction multiple times during the learning process. The bottom-clause construction algorithm queries the database multiple times each of which selects all tuples in a table that match given constants from the training data. We leverage RDBMS indexing to improve the running time of these queries.

Stored Procedures: We implement the bottom-clause construction algorithm inside a stored procedure to reduce the number of API calls made from Castor to

the RDBMS. Castor makes only one API call per each bottom-clause. The first time that Castor is run on a schema, it creates the stored procedure that implements the bottom-clause construction algorithm for the given schema. Castor reuses the stored procedure when the algorithm is run again, with either new training data or updated database instance.

Efficient Clause Evaluation: One approach to computing the number of positive (negative) examples covered by a clause is to join the table containing the positive (negative) examples with the tables corresponding to all literals in the body of the clause. If two literals share a variable, then a natural join between the two columns corresponding to the shared variable in the literals is used. This strategy works well when clauses are short, as in top-down algorithms [31]. 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 [19]. Resumer2 efficiently checks if clause C covers example e by deciding the subsumption between C and the ground bottom-clause $\bot_e$ of e. Given clause C and a set of examples E, Castor checks if C covers each $e \in E$ separately. Castor divides E in subsets and performs coverage testing for each subset in parallel.

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

Minimizing Clauses: Bottom-up algorithms such as Castor produce large clauses, which are expensive to evaluate. Castor minimizes bottom-clauses by removing syntactically redundant literals. A literal L in clause C is *redundant* if C is equivalent to $C' = C - \{L\}$. Clause equivalence between C and C' can be determined by checking whether C θ-subsumes C' and C' θ-subsumes C. Castor minimizes clauses using theta-transformation [8]. It uses a polynomial-time approximation of the clausal-subsumption test, which is efficient and retains the property of correctness. Given clause C, for each literal L in C, the algorithm checks if $C \subseteq C' = C - \{L\}$. If this holds, then L is redundant and will be removed. Minimizing bottom-clauses reduces the hypothesis space considered by Castor. It also makes coverage testing faster. Castor also minimizes learned clauses before adding them to the definition. This ensures that clauses are concise and interpretable.

8 Query-based algorithms

In this section, we consider query-based learning algorithms, which learn exact definitions by asking queries to an oracle [18,27,4,2]. This type of algorithms have been recently used in various areas of database management, such as finding schema mappings and designing usable query interfaces [6,2]. Queries can be of multiple types, however the most common types are equivalence queries and membership queries. In equivalence queries (EQ), the learner presents a hypothesis to the oracle and the oracle returns *yes* if the hypothesis is equal to the target relation definition, otherwise it returns a counter-example. In membership queries (MQ), the learner asks if an example is a positive example, and the oracle answers *yes* or *no*.

Because query-based algorithms follow a different learning model, Definition 3 is not suited for evaluating their schema (in)dependence. Since a query-based algorithm can ask the oracle whether candidate definitions are correct, the algorithm will always learn the correct definitions by asking sufficient number of queries from the oracle. As it takes time and/or resources to answer queries, a desirable query-based algorithm should not ask too many queries [4]. For instance, some database query interfaces use query-based algorithms to discover users' intents [2]. Because the oracle for these algorithms is the user of the database, a more desired algorithm should figure out the user's intent by asking fewer queries from the user.

Query-based algorithms are theoretically evaluated by their *query complexity* – the asymptotic number of queries asked by the algorithm [18]. Therefore, we analyze the impact of schema transformations on the query complexity of these algorithms. Generally, if an algorithm has different asymptotic behavior over equivalent schemas, then the algorithm is schema dependent. One way to show that an algorithm has different asymptotic behavior over different schemas is by comparing the lower bound on the query complexity of the algorithm against the upper bound on its query complexity. If the lower bound under one of the schemas is greater than the upper bound under another schema, then the algorithm is highly schema dependent. Of course, this is not a desirable property, as this means that the choice of representation has a huge impact on the performance of

the algorithm. However, we prove that a popular query-based algorithm called $A2$ suffers from this property.

$A2$ [18] is a query-based learning algorithm that learns function-free, first-order Horn expressions. The reasons for choosing this algorithm are three fold: i) $A2$ is representative of query-based learning algorithms that work on the relational model, ii) there is an implementation of the algorithm [4], iii) $A2$ is a generalization to the relational model of a classic query-based propositional algorithm [3].

Let $p_{\mathcal{R}}$ be the number of relations in schema $\mathcal{R}$ and $a_{\mathcal{R}}$ be the largest arity of any relation in schema $\mathcal{R}$. Let k be the largest number of variables in a clause, m be the number of clauses in the definition of the target relation, and n be the largest number of constants (i.e. objects) in any example. Parameters k, m, and n are independent of the schema. The upper bound on the number of EQs and MQs made by the $A2$ algorithm over schema $\mathcal{R}$ is $O(m^2(p_{\mathcal{R}})k^{(a_{\mathcal{R}})+3k}+nm(p_{\mathcal{R}})k^{(a_{\mathcal{R}})+k})$, and the lower bound is $\Omega(m(p_{\mathcal{R}})k^{(a_{\mathcal{R}})})$ [18].

Theorem 5 *There is a schema $\mathcal{R}$ and decomposition τ, where $\tau(\mathcal{R}) = \mathcal{S}$, such that* $\Omega(m(p_{\mathcal{R}})k^{(a_{\mathcal{R}})}) > O(m^2(p_{\mathcal{S}})k^{(a_{\mathcal{S}})+3k}+nm(p_{\mathcal{S}})k^{(a_{\mathcal{S}})+k})$.

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

Let $\mathcal{L}$ be the hypothesis language that consists of the subset of Horn definitions that contain a single clause in which no self-joins are allowed. All definitions in $\mathcal{L}$ under schema $\mathcal{R}$ have the form $T(\mathbf{u}) \leftarrow R(x_1, x_2, \cdots, x_l)$, where T is the target relation and $\mathbf{u}$ is a subset of $\{x_1, x_2, \cdots, x_l\}$.

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

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

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

where the left side of the inequality is the lower bound on the query complexity under schema $\mathcal{R}$ and the right side is the upper bound on the query complexity under schema $\mathcal{S}$. The operator $>$ means that $A2$ will always ask asymptotically more queries under schema $\mathcal{R}$ than under schema $\mathcal{S}$. We have that k and m are the same for both schemas. We can also ignore n as it is independent of the hypothesis space and the schemas. Therefore, by canceling out some terms, the previous inequality can be rewritten as

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

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

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

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

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

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

The lower bound of $A2$ is the Vapnik-Chevonenkis dimension (VC-Dim) of the hypothesis language that consists of function-free, first-order Horn expressions. Therefore, we have proven in Theorem 5 that there are cases where the lower bound on the query complexity of *any* algorithm under this hypothesis language is greater than the upper bound on the query complexity of $A2$. This means that any algorithm that is as good as $A2$ (does not ask more queries than $A2$) is highly dependent on the schema details.

9 Experiments

9.1 Experimental Settings

We use three datasets whose statistics are shown in Table 2. The **HIV-Large** dataset contains information about 42,000 chemical compounds obtained from the National Cancer Institute's AIDS antiviral screen (*wiki.nci.nih.gov/display/NCIDTPdata*). The schema contains relation *compound(comp,atm)*, which indicates that compound *comp* contains atom *atm*. It also has relations that indicate the chemical element that an atom represents, e.g., *element_C(atm)*, as well as relations to indicate properties of each atom, e.g., *p2_1(atm)*. The schema represents a bond between two atoms by relation *bonds(bd,atm1,atm2)*, and it has a relation for each type of a bond, e.g., *bondType1(bd,t1)*. The goal is to learn the relation *hivActive(compound)*, which indicates that *compound* has anti-HIV activity. The original HIV dataset is stored in flat files and does not have any information about its constraints. We explored the database

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.

for possible dependencies. 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 imposed INDs can be found in [24]. 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 [28]. The goal is to 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. 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 added added INDs to the schema. Details about the original and additional INDs can be found in [24]. We enforce the constraints by removing a small fraction of tuples, 159 tuples, from the original dataset. We transform the original schema to three other different schemas. The original and a composed schema, called 4NF, are shown in Table 1. We compose *courseLevel* and *taughtBy* relations in 4NF schema to create a more denormalized schema, named Denormalized-1, and compose *courseLevel*, *taughtBy*, and *professor* in 4NF schema to generate the fourth schema, named Denormalized-2.

The **IMDb** (*imdb.com*) dataset contains information about movies. We learn the target relation *dramaDirector(director)*, which indicates that *director* has directed a drama movie. JMDB (*jmdb.de*) provides a relational database of IMDb data under a 4NF schema. We create a subset of JMDB database by selecting the movies produced after year 2000 and their related entities, e.g., actors, directors, producers. The relationships between relation *movie(id,title,year)* and its related relations, e.g., *director(id,name)*, are stored in relations *movies2X* where X is the name of the related entity set, e.g., *movies2director(id,directorid)*. The resulting database has 11 INDs with equality in the form of *movies2X[Xid] =X[id]*. To test over more transformations, we have changed 5 INDs in the form of subset to INDs in the form of equality, e.g., *movies2X[id]* $\subseteq$ *movie[id]* to *movies2X[id] = movie[id]*, by removing some tuples from the database. We use the first set of 11 INDs with equality to compose 11 pairs of relations in JMDB schema to create a new schema, called Denormalized. We use the second set of INDs with equality to compose 5 relations in JMDB schema, and create a schema called Stanford that follows a structure similar to the one used in the Stanford Movie DB (*infolab.stanford.edu/pub/movies*). The three schemas and the full list of INDs in IMDb data can be found in [24]. In the UW-CSE and IMDb datasets, we generate negative examples by using the closed-world assumption, and then sample to obtain twice as many negative examples as positive examples.

We compare Castor to three relational learning systems: FOIL [26], Aleph [29], and GILPS [23]. **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 [21]. To differentiate the two variations of Aleph used in our experiment, we call the default implementation of Aleph **Aleph-Progol**. GILPS implements **ProGolem**, which is a bottom-up algorithm.

Aleph contains the parameter *clauselength*, which restricts the size of the learned clauses. Over HIV-Large and HIV-2K4K, the definition for the target relation must contain long clauses. With the default value of *clauselength* = 4, Aleph-FOIL and Aleph-Progol do not learn any clause. Therefore, we set this parameter to have values of 10 and 15. Details about parameters used in all systems can be found in [24].

There are far fewer query-based relational learning systems available than the ones that use samples for learning. To empirically evaluate the schema independence of query-based learning methods, we use the **LogAn-H** system [4], which is an implementation of the A2 algorithm [18].

We compare the quality of the learned definitions using the metrics of *precision* 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. Precision and recall

are between 0 and 1, where an ideal definition delivers both precision and recall of 1. Similar to other machine learning tasks, it is not often possible to learn an ideal definition for a target concept due to various reasons, such as the hardness of the target concept 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 be important in a financial application but vital in a medical application. Nevertheless, definitions with higher precision and/or recall are generally more desirable [26, 23, 29]. 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.

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

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

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

9.2 Sample-based Algorithms

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

HIV datasets. Aleph-FOIL, Aleph-Progol and Castor are the only algorithms that scale to the HIV-2K4K dataset. Aleph-FOIL and Castor also scale to the HIV-Large dataset. The definitions learned by Aleph-FOIL and Aleph-Progol over different schemas are not equivalent as shown by their precision and recall values across schemas in Table 3. Different schemas cause Aleph-FOIL and Aleph-Progol to explore different regions of the hypothesis space. Aleph-FOIL and Aleph-Progol are not able to find any definition over the 4NF-2 schema of HIV-Large and HIV-2K4K datasets. The reason is that any good clause must contain information about bonds. In the 4NF-2 schema, this information is represented by two relations, *bondSource* and *bondTarget*, and three more to indicate their types. With a top-down search, these algorithms are not able to find a clause that contains these relations. Aleph-FOIL terminates without learning anything and Aleph-Progol does not terminate after 75 hours. Aleph-Progol does not terminate after 75 hours over the 4NF-2 schema of HIV-2K4K. FOIL crashes on both HIV datasets. ProGolem does not learn anything after 5 days running, even on smaller subsets of the HIV dataset.

UW-CSE dataset. As shown in Table 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. 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. 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

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

Table 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 (m)	6.4	1,229	476.4
Aleph-Progol	Precision	0.66	1	0.69
	Recall	0.47	1	0.52
	Time (m)	312.9	1,248	937.4
Castor	Precision	1	1	1
	Recall	1	1	1
	Time (m)	15.14	108.15	32.4

Table 5 Results of learning relations over IMDb data.

different schemas are largely different.

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

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

Aleph-FOIL and Castor are the only algorithms that scale to the HIV-Large dataset. Aleph-FOIL with *clauselength* = 10 is more efficient than Castor. However, when *clauselength* is set to 15, it becomes less efficient, as shown in Table 3. Aleph-FOIL with both *clauselength* = 10 and 15 is also faster than Castor over the HIV-2K4K dataset. In general, top-down algorithms that follow greedy search strategies are expected to be more efficient than bottom-up algorithms. Top-down algorithms have a search bias for shorter clauses, which are cheaper to compute. They usually limit the maximum length of the clauses to be learned. Further, algorithms that follow greedy search strategies can be more efficient. This is exploited by related work that focuses on efficiency [31,14]. 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 7.5. 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. 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

HIV-2K4K			
Metric	Initial	4NF-1	4NF-2
Precision	0.77	0.79	0.73
Recall	0.63	0.87	0.76
Time (m)	27	14.8	576

UW-CSE				
Metric	Original	4NF	Denorm-1	Denorm-2
Precision	0.93	0.93	0.93	0.93
Recall	0.54	0.54	0.54	0.54
Time (s)	8	8.9	9.1	13.3

IMDb			
Metric	JMDB	Stanford	Denormalized
Precision	1	0.98	1
Recall	1	0.84	1
Time (m)	7.3	90.8	8.1

Table 6 Results of Castor learning relations over HIV-2K4K, UW-CSE and IMDb data using INDs in the form of subset.

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

The efficiency of Castor is comparable to that of Aleph-FOIL and Aleph-Progol over the Original and 4NF schemas of the UW-CSE dataset. The running time of Aleph-FOIL and Aleph-Progol is heavily 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.

General decomposition/ composition. As it is explained in Section 7.4, there are two methods to achieve robustness over the schema variations created by the INDs in general forms. One can use a preprocessing step to check whether the IND holds in the form of equality over the available instance. Then, one can apply the original Castor algorithm and achieve complete schema independence. The empirical results of this method are exactly the same as the ones of the original Castor algorithm with the overhead of its preprocessing step. Another method is to use the INDs in general form directly without any preprocessing. We empirically evaluate the robustness of the latter method in this section. To explore general 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 9.1 for these datasets are general decomposition/ composition and not bijective. We run the extended version of Castor from Section 7.4 using the aforementioned INDs and all other regular INDs in each schema. Table 6 shows the results of Castor learning relations over the HIV-2K4K, UW-CSE and IMDb datasets, using only INDs in the form of subset. The extension of Castor gets the same results as in Table 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: it delivers precision of 0.77, 0.79, and 0.73 and recall of 0.63, 0.87, and 0.76 over the Initial, 4NF-1, and 4NF-2 schemas, respectively. This is because it cannot access some tuples in the bottom-clause construction in these databases as explained in Section 7.4. Its precisions are equal or higher than the those of Aleph-FOIL and Aleph-Progol over all schemas and its recall is higher than that of Aleph-FOIL and Aleph-Progol in 4NF-2 schema. But, its recall is lower than the recall of Aleph-FOIL and Aleph-Progol over the Initial and Aleph-Progol over 4NF-1 schemas.

9.3 Impact of Castor Design Choices

We evaluate the impact of parallelization and the use of stored procedures on Castor's running time. There are some variations in the running times of Castor compared to the experiments in the previous section. This is because we run experiments again, and the running times may fluctuate.

Impact of parallelization. Castor performs coverage tests in parallel to improve its running time. Figure 1 shows the impact of parallelization on Castor's running time over HIV-Large (Initial schema), HIV-2K4K (Initial schema) and IMDb (JMDB schema). Over both HIV-Large and HIV-2K4K datasets, Castor benefits from parallelization. Over the HIV-Large dataset, the best performance is obtained by using 32 threads, which reduces the running time by half compared to using 1 thread. Over the HIV-2K4K dataset, the running time also reduces significantly with parallelization

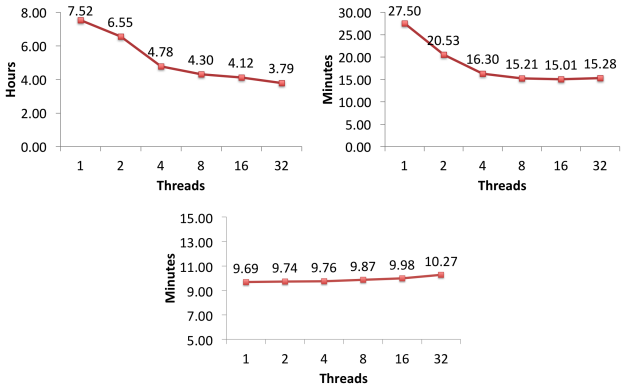

Fig. 1 Impact of parallelization on Castor's running time over the HIV-Large (top-left), HIV-2K4K (top-right), and IMDb datasets (bottom).

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

Table 7 Impact of stored procedures on Castor's running time over the HIV-Large, HIV-2K4K, and IMDb datasets

and the best performance is obtained with 16 threads. Over the IMDb dataset, there is no benefit in using parallelization. This is because Castor does not need to perform many coverage tests, as it is able to find the perfect definition very quickly. In this case, most of Castor's running time is spent in creating the ground bottom-clauses, as explain in Section 7.5. Because the UW-CSE dataset is very small, there is no need for parallelization. Notice that sequential Castor (1 thread) is more efficient than Aleph-FOIL with *clauselength* = *15* over the HIV-Large dataset and more efficient than Aleph-Progol over the HIV-2K4K and IMDb datasets. This shows that besides parallelization, the techniques explained in Section 7.5 allow Castor to run efficiently.

Impact of using stored procedures. Castor uses the bottom-clause construction algorithm to generate bottom-clauses in the *LearnClause* procedure, as well as to generate ground bottom-clauses, used to test coverage. As mentioned in Section 7.5, we implement the bottom-clause construction algorithm inside a stored procedure. To evaluate the benefit of using stored procedures, we also implement a version of Castor that does not use stored procedures. Table 7 shows the running time of the versions of Castor with and without stored procedures over the HIV-Large (Initial schema), HIV-2K4K (Initial schema) and IMDb (JMDB schema) datasets. The version of Castor that uses stored procedures obtains between 1.25x and 1.9x speedup over the version that does not use stored procedures.

9.4 Query-based Algorithms

We used the interactive algorithm with automatic user mode in the LogAn-H system. In this mode, the system is told the Horn definition to be learned, so that it can act as an oracle. Then the algorithm's queries are answered automatically until it learns the exact definition. When answering EQs, the counter-examples are produced by the system. Therefore, LogAn-H only takes as input the schema of the dataset, but not the database instance. We performed experiments using the schemas of the UW-CSE dataset. We generated random Horn definitions over the Denormalized-2 schema of the UW-CSE dataset. The definition generator has a parameter to indicate the number of variables in each clause. To generate the head of each clause, we created a new relation of random arity, where the minimum arity is 1 and the maximum arity is the maximum arity of the relations in the Denormalized-2 schema. The body of each clause can be of any length as long as the number of variables in the clause is equal to the specified parameter and all variables appearing in the head relation also appear in any relation in the body. The body of the clause is composed of randomly chosen relations, where each relation can be the head relation or any relation in the input schema. Head and body relations are populated with variables, where each variable is randomly chosen to be an existing or new variable.

After generating each random Horn definitions over the Denormalized-2 schema, we transformed these expressions to the Denormalized-1, 4NF and Original schemas by simply doing vertical decomposition to each of the clauses in a definition. We varied the number of clauses in a definition to be between 1 and 5, each containing between 4 and 8 variables. Therefore, we generated 50 random definitions for each setting. We ran the LogAn-H system and recorded the number of queries required to learn each definition under each schema. The number of EQs and MQs asked by the algorithm is presented in Figure 2. The average number of EQs required by the A2 algorithm is constant for different number of variables and similar throughout all schemas. However, this is not the case for MQs. Particularly, the number of MQs is greater for more decomposed schemas, e.g., Original schema. Further, the number of MQs also increases with the number of variables. This difference of MQs between the schemas originates from a step in the A2 algorithm that removes non-essential literals in ground bottom-clauses generated from negative examples. This process is similar to Castor's negative reduction. It removes a literal and asks an MQ to verify whether the example is still negative. Therefore, the

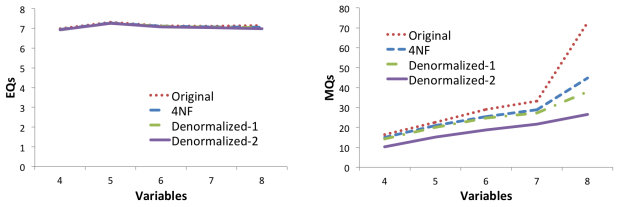

Fig. 2 Average number of equivalence (left) and membership (right) queries for the A2 algorithm.

more decomposed the schema is, the more literals can be removed, hence more MQs are asked.

10 Conclusion

We defined the property of schema independence for relational learning algorithms, which states that the output of these algorithms should not depend on the schema used to represent their input databases. We proved that current well-known relational learning algorithms are not schema independent over composition/decomposition. We proposed a new algorithm, Castor, that leverages schema constraints to achieve schema independence. Our empirical results on benchmark and real datasets validated our theoretical results and showed that Castor is efficient and more or as effective as current relational learning algorithms.

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. D. Angluin, M. Frazier, and L. Pitt. Learning conjunctions of Horn clauses. *Mach. Learn.*, 9(2-3):147–164, 1992.
4. M. Arias, R. Khardon, and J. Maloberti. Learning Horn expressions with LOGAN-H. *J. Mach. Learn. Res.*, 8:549–587, 2007.
5. P. Atzeni, G. Ausiello, C. Batini, and M. Moscarini. Inclusion and Equivalent Between Relational Database Schemata. *TCS*, 1982.
6. B. T. Cate, V. Dalmau, and P. G. Kolaitis. Learning schema mappings. *TODS*, 38(4):28:1–28:31, 2013.
7. Y. Chen, S. Goldberg, D. Z. Wang, and S. S. Johri. Ontological pathfinding : Mining first-order knowledge from large knowledge bases. In *SIGMOD*, 2016.
8. 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, 2003.
9. J. Davis, E. S. Burnside, I. de Castro Dutra, D. Page, R. Ramakrishnan, V. S. Costa, and J. W. Shavlik. View learning for statistical relational learning: With an application to mammography. In *IJCAI*, 2005.
10. L. De Raedt. *Logical and Relational Learning*. Springer Publishing Company, Incorporated, 1st edition, 2010.
11. R. Fagin. Inverting schema mappings. *TODS*, 32(4), 2007.
12. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.
13. W. Fan and P. Bohannon. Information Preserving XML Schema Embedding. *TODS*, 33(1), 2008.
14. L. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. In *VLDB Journal*, 2015.
15. L. Getoor and A. Machanavajjhala. Entity resolution in big data. In *KDD*, 2013.
16. L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.
17. R. Hull. Relative Information Capacity of Simple Relational Database Schemata. In *PODS*, 1984.
18. R. Khardon. Learning function-free Horn expressions. *Machine Learning*, 37(3):241–275, 1999.
19. O. Kuželka and F. Železný. A restarted strategy for efficient subsumption testing. *Fundam. Inf.*, 89(1):95–109, 2009.
20. H. Li, C.-Y. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13), 2015.
21. S. Muggleton. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13:245–286, 1995.
22. S. Muggleton and C. Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.
23. S. Muggleton, J. C. A. Santos, and A. Tamaddoni-Nezhad. Progolem: A system based on relative minimal generalisation. In *ILP*, volume 5989, 2009.
24. J. Picado, A. Termehchy, A. Fern, and P. Ataei. Schema independent relational learning. http://arxiv.org/abs/1508.03846, 2015.
25. J. Picado, A. Termehchy, A. Fern, and P. Ataei. Schema Independent Relational Learning. In *SIGMOD*, 2017.
26. J. R. Quinlan. Learning Logical Definitions From Relations. *Machine Learning*, 5, 1990.
27. C. Reddy and P. Tadepalli. Learning Horn definitions: Theory and an application to planning. *New Generation Computing*, 17:77–98, 1998.
28. M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, Feb. 2006.
29. A. Srinivasan. *The Aleph Manual*, 2004.
30. A. Termehchy, M. Winslett, and Y. Chodpathumwan. How Schema Independent Are Schema Free Query Interfaces? In *ICDE*, 2011.
31. Q. Zeng, J. M. Patel, and D. Page. QuickFOIL: Scalable inductive logic programming. *PVLDB*, 2014.