Usable & Scalable Learning Over Relational Data With Automatic Language Bias

ABSTRACT

Relational databases are valuable resources for learning novel and interesting relations and concepts. In order to constraint the search through the large space of candidate definitions, users must tune the algorithm by specifying a language bias. Unfortunately, specifying the language bias is done via trial and error and is guided by the expert's intuitions. We propose AutoBias, a system that leverages information in the schema and content of the database to automatically induce the language bias used by popular relational learning systems. We show that AutoBias delivers the same accuracy as using manually-written language bias by imposing only a slight overhead on the learning time.

ACM Reference Format:
. 2021. Usable & Scalable Learning Over Relational Data With Automatic Language Bias. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A large body of machine learning and AI is focused on learning models composed of a set of (probabilistic) logical rules over relational databases and knowledge bases [6, 12, 13, 15, 21, 28, 31, 33, 45, 47, 54, 55]. Consider the UW 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. One may want to predict the new relation *advisedBy(stud,prof)*, which indicates that the student *stud* is advised by professor *prof.* Given the UW database and positive and negative training examples of the *advisedBy* relation, *relational learning algorithms* exploit the relational structure of the data to find a *definition* of this relation in terms of the other existing relations in the database [13, 15, 28, 31, 33, 43, 45, 55, 57]. Learned definitions are usually (probabilistic) first-order logic formulas and often restricted to Datalog programs. For example, a relational learning algorithm may learn the following Datalog program for the *advisedBy* relation:

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

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

Relational models offer several key advantages over other learning models on relational data [6, 15, 31, 45]. First, non-relational learning methods, e.g., logistic regression, rely on the assumption that the data points are independent and and follow an identical distribution (IID) [35]. It is well established that the IID assumption is often violated over relational data, therefore, using non-relational models may result in models that are too biased to the training data and have low accuracy over the test data [13, 15, 21, 45]. This has significantly limited the application of non-relational models over relational data to very restricted cases, e.g., a single fact table and potentially some dimension tables with one-to-many relationships and IID distributions [23, 29, 45]. Second, relational models are interpretable and easy to understand. Third, as relational models directly leverage the structure of the data, users do not need to perform lengthy and cumbersome process of feature engineering. As a matter of fact, when users choose to use non-relational models over structured data, they often use relational learning methods to learn and find features for their models [24, 31]. Relational models are widely used in AI, e.g., information extraction [15, 28, 45, 53], question answering [39, 40, 45, 55]; data management, e.g., usable query interfaces [3, 4, 8, 17, 26, 27, 32, 49], entity resolution [7, 16, 20], schema mapping discovery [9, 50–52], data cleaning [5, 25, 46], data provenance [10, 14], schema discovery [7]; and software engineering [22].

The space of possible hypotheses that a relational learning algorithm should explores consists of all Datalog programs defined over the schema of the underlying data, which is enormous over a large database [6, 12, 43, 45, 57]. Therefore, users constraint the hypothesis space of the algorithm using a *language bias.* In particular, users leverage their knowledge about the underlying data and target concept to restrict the relations, the connections and join paths between the relations, and the constants and values used in the hypotheses to ensure that the hypothesis space is both sufficiently small and contains all promising definitions, which makes effective and efficient learning possible. Relational learning

Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

systems usually accept the language bias in form of a set of declarative rules [13]. To develop the language bias that facilitates effective and efficient learning, a user should both know the internals of the learning algorithm and the schema and content of the input database and have a relatively clear intuition on the structure of accurate definitions for the target concept. However, most end users and domain experts that use relational learning may not be familiar with the database concepts, such as schema, or writing declarative rules. Moreover, due to the huge volume, complex structures, or potentially frequent evaluations of most datasets, it is challenging for users to know structure and content of the underlying data sufficiently well to specify language biases that deliver accurate models efficiently. Also, users often do not have the sufficient understanding of the target concept to describe precisely a sufficiently small and promising hypothesis space, particularly for difficult target concepts, e.g., drug compounds that help treating viral infections [43, 57]. Furthermore, language bias over for learning over a large dataset may consist of hundreds of declarative rules, which are developed via tedious process of trial and error and hard to maintain in face of data evolution. In our conversations with (statistical) relational learning experts, they have called language bias development the "black magic" needed to make relational learning work. Generally, language bias is deemed to be a main challenge for learning over structured data [6].

In this paper, we propose a novel approach that leverages the information in the schema and content of the database to generate language bias automatically with minimal user intervention. Our method uses the exact and approximate database constraints to find promising patterns in the data. These constraints are usually available in the schema of the database or can be discovered from the database instance [1, 19, 42]. This method does not generally limit the space of the search for the learning algorithm as tightly as the ones written manually by the experts, therefore, it may result in extremely time-consuming learning over large databases. To address this challenge, we leverage the literature of sampling techniques over relational data and propose novel sampling methods over the hypothesis space induced by the aforementioned language bias. Our proposed techniques enable the learning algorithms to explore a sufficiently small and diverse hypotheses to learn accurate definitions efficiently. Since the space of explored hypotheses may still be too large for many datasets, we also propose sampling methods to ensure the learning algorithm quickly determine the quality of every chosen hypothesis in the sampled space. More specifically, our contributions in this paper are:

- We introduce the problem of setting the language bias for relational learning automatically.
- We propose a new system called *AutoBias*, which leverages the exact and approximate database constraints [2] and content to induce language bias automatically (Section 4).
- Since the language bias induced by our method may not sufficiently restrict the hypothesis space over large datasets, we propose random sampling techniques to enable the learning algorithm explore the hypothesis space efficiently. However, it takes an extremely long time to construct and materialize all Datalog definitions in the hypothesis space before selecting a random sample of them. Our method provides a random sample of the hypothesis space by materializing only the randomly chosen definitions. Randomly sampled definitions might be biased toward highly connected tuples in the data, which may reduce the accuracy of learned model for large and diverse datasets. Thus, we also propose a stratified sampling method that delivers more diverse hypothesis than random (Section 5.2).
- After choosing a hypothesis, a relational learning algorithm has to evaluate the quality of the hypothesis, i.e., whether it covers sufficiently many positive and few negative examples. Since each hypothesis may contain hundreds of literals, e.g., joins of hundreds of relations, it takes a very long time to evaluate its quality over large data. We propose sampling techniques that effectively evaluate the quality of each hypothesis efficiently (Section 6).
- We empirically evaluate our proposed methods over real-world and large databases. Our empirical study indicates that our proposed language bias generation method delivers almost as accurate models as the ones developed by experts over multiple datasets. They also show that random sampling approach improves the efficiency of our system significantly and delivers more effective or as effective results than the state-of-the-art sampling techniques over large databases.

2 BACKGROUND

2.1 Basic Definitions

An *atom* is a formula in the form of $R(e_1, \ldots, e_n)$, where R is a relation symbol. A *literal* is an atom, or the negation of an atom. Each attribute in a literal is set to either a variable or a constant, i.e., value. Variable and constants are also called *terms*. A *Horn clause* (clause for short) is a finite set of literals that contains exactly one positive literal called *head-literal*. Horn clauses are also called conjunctive queries. A *Horn definition* is a set of Horn clauses with the same head-literal.

A relational learning algorithm learns a Horn definition from input relational databases and training data. The learned definition is called the hypothesis, which is usually restricted to non-recursive Datalog definitions without negation, i.e., unions of conjunctive queries, for efficiency reasons. The *hypothesis space* is the set of all candidate Horn definitions that

Table 1: Schema for the UW dataset.

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

Table 2: A subset of predicate and mode definitions for the UW dataset.

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

the algorithm can explore. Each member of the hypothesis space is a *hypothesis*. Given a database instance I, clause C *covers* example e if $I \wedge C \models e$, where $\models$ is the entailment operator, i.e., if I and C are true, then e is true. Definition H covers an example e if at least one its clauses covers e. Relational learning algorithms search over the hypothesis space for a definition that covers as many positive examples as possible, while covering the fewest possible negative examples.

2.2 Language Bias

In relational learning, language bias restricts the structure and syntax of the generated clauses. It is specified through predicate and mode definitions [13].

2.2.1 Predicate Definitions. Predicate definitions assign one or more *types* to each attribute in a database relation. In a candidate clause, two relations can be joined over two attributes (i.e., attributes are assigned the same variable) only if the attributes have the same type. For instance, in Table 2, the predicate definition `student(T1)` indicates that the attribute in relation *student* is of type `T1`, and the predicate definition `inPhase(T1,T2)` indicates that the first and second attributes of relation *inPhase* are of type `T1` and `T2`, respectively. Hence, relations *student* and *inPhase* can be joined on attributes *student[stud]* and *inPhase[stud]*. Multiple types may be assigned to an attribute. For example the predicate definitions `publication(T5,T1)` and `publication(T5,T3)` indicate that the attribute *author* in relation *publication* belongs to both types `T1` and `T3`. Predicate definitions restrict the joins that appear in a candidate clause: two relations are joined only if their attributes share a type.

Intuitively, predicate definitions should assign the same types to attributes that refer to entities of the same *semantic type*. For instance, attributes *student[stud]* and *inPhase[stud]* both refer to the entity type *student*. Therefore, predicate definitions should assign the same type to these attributes. On the other hand, attribute *inPhase[phase]* refers to entities of type *phase*. Therefore, this attribute should be of a different type. Note that relying on attribute names would not be a reliable way to inferring the semantic types of entities stored in an attribute. A user should know the schema of the database and the meaning of all attributes in order to write effective predicate definitions.

2.2.2 Mode Definitions. Mode definitions indicate whether a term in an literal should be a new variable, i.e., existentially quantified variable, an existing variable, i.e., appears in a previously added literal, or a constant. They do so by assigning one or more symbols to each attribute in a relation. *Symbol* + indicates that a term must be an existing variable. *Symbol* − indicates that a term can be an existing variable or a new variable. For instance, the mode definition `inPhase(+,-)` in Table 2 indicates that the first term must be an existing variable and the second term can be either an existing or a new variable. *Symbol* # indicates that a term should be a constant. For instance, the mode definition `inPhase(+,#)` indicates that the second term must be a constant.

Mode definitions restrict the candidate clauses that are explored by the learning algorithm. Each literal in a candidate clause must satisfy at least one mode definition. Some mode definitions do not add any value to the creation of candidate clauses. For instance, mode definition `inPhase(+,+)` means that both variables in a new literal must be existing variables. The same literal can be created from mode definitions `inPhase(+,-)` or `inPhase(-,+)`. Therefore, mode definition `inPhase(+,+)` does not add new more information to the candidate clause. On the other hand, mode definition `inPhase(-,-)` means that both variables in a literal must be new variables. In this case, the new literal would not be connected to any previously added literal, resulting in a Cartesian product in the clause. A user should know the learning algorithm and have an intuition of the desired hypotheses in order to write effective mode definitions. We explain how predicate and mode definitions are used in the learning algorithm in Section 3.1.

3 AUTOBIAS LEARNING ALGORITHM

AutoBias uses the same learning algorithm as existing relational learning algorithms [45]. In this section, we explain this algorithm and how it uses language bias to learn efficiently. Similar to other relational learning algorithms, AutoBias is a sequential covering algorithm [33, 37, 43, 45, 53, 57]. Algorithm 1 depicts a general sequential covering algorithm. The algorithm constructs one clause at a time using the *LearnClause* function. If the clause satisfies the minimum criterion, it adds the clause to the learned definition and discards the positive examples covered by the definition. It stops when all positive examples are covered by the definition.

There are two main approaches in developing the *LearnClause* function. In the *top-down* approach, the algorithm starts with an empty definition and iteratively add literals to the definition until the definition cannot be improved [53, 55, 57]. In the bottom-up approach, the algorithm first finds relevant patterns in the data and then generalizes them to find clauses that capture the training examples accurately [34, 37, 43]. AutoBias follows the latter approach. Under this approach, the *LearnClause* function contains two main steps. In the first step, it constructs the most specific clause that covers a positive example, relative to the database. This clause is called the *bottom-clause*. In the second step, it generalizes the bottom-clause to cover more positive examples, while covering the fewest possible negative examples. It is shown that the bottom-up approach usually delivers more effective results than those of top-down methods [13, 43].We now explain each step in more detail.

Algorithm 1: Sequential covering algorithm.

```
Input  : Database instance $I$, positive examples $E^+$,
         negative examples $E^-$
Output : A Horn definition $H$
1 $H = \{\}$
2 $U = E^+$
3 while $U$ is not empty do
4     $C = LearnClause(I, U, E^-)$
5     if $C$ satisfies minimum criterion then
6         $H = H \cup C$
7         $U = U - \{e \in U | H \wedge I \models e\}$
8 return $H$
```

3.1 Bottom-clause Construction

A *bottom-clause* C_e associated with an example e is the most specific clause in the hypothesis space that covers e relative to the underlying database I. The bottom-clause construction algorithm consists of two phases. 1) Find all the information in I relevant to e. The information relevant to example e is the set of tuples $I_e \subseteq I$ that are connected to e. 2) Given I_e, create the bottom-clause C_e. The bottom-clause construction algorithm is depicted in Algorithm 2. Relational learning algorithms use predicate and mode definitions to restrict the structure and syntax of the bottom-clauses [13, 43]. We now explain the bottom-clause construction and how it uses predicate and mode definitions.

Assume that we want to create the bottom-clause for example e, relative to database I. The algorithm maintains a hash table that maps constants to variables. The algorithm first assigns new variables to constants in example e, and inserts the mapping from constants to variables in the hash table. It creates the head of the bottom-clause by replacing the constants in e with their assigned variables. Then, for each constant a in the hash table, the algorithm looks for relations that contain attributes with the same type as a and then searches for tuples in these relations that contain constant a. The type of a constant is determined by the attribute in which the constant appears. The attribute types are assigned using **predicate definitions**. To further restrict the search, the algorithm only considers attributes which contain symbol +, according to the **mode definitions**. For each tuple, the algorithm creates one or more literals with the same relation name as the tuple and adds the literals to the body of the bottom-clause. The algorithm also uses mode definitions to determine whether an attribute in a literal should be a variable or a constant. An attribute A in relation R can be a variable if the mode definitions for relation R contain symbols + or − on attribute R. Attribute A can be a constant if the mode definitions for relation R contain symbols # on attribute R. If an attribute A can be both a variable and a constant, the algorithm creates two new literals, one literal containing a variable in attribute A and the containing a constant in attribute A. If an attribute should be a variable according to mode definitions, and the constant in this attribute is new, the algorithm assigns a new variable to the constant and adds the new mapping to the hash table. In the following iterations, the algorithm selects tuples in the database that contain the newly added constants to the hash table and adds their corresponding literals to the clause. It finishes after the user-specified number of iterations.

Algorithm 2: Bottom-clause construction.

```
Input  : example $e$, # of iterations $d$, sample size $s$
Output : bottom-clause $C_e$
1 $I_e = \{\}$
2 $M = \{\}$ // $M$ stores known constants
3 add constants in $e$ to $M$
4 for $i = 1$ to $d$ do
5     foreach relation $R \in I$ do
6         foreach attribute $A$ in $R$ do
7             $I_R = \sigma_{A \in M}(R)$
8             foreach tuple $t \in I_R$ do
9                 add $t$ to $I_e$ and constants in $t$ to $M$
10 $C_e$ = create clause from $e$ and $I_e$
11 return $C_e$
```

Example 3.1. Consider the database I in Table 3, the predicate and mode definitions in Table 2, and a positive example e = *advisedBy(alice,bob)*. Given that the user sets the number of iterations d to 1, the bottom-clause associated with e and relative to I is:

$$advisedBy(x, y) \leftarrow student(x), professor(y),$$
$$inPhase(x, u), inPhase(x, post\_quals), hasPosition(y, v),$$
$$publication(z, x), publication(z, y).$$

Table 3: Fragments of the UW database.

student(alice)	professor(bob)
student(john)	professor(mary)
inPhase(alice,post_quals)	hasPosition(bob,assistant_prof)
inPhase(john,post_quals)	hasPosition(mary,associate_prof)
publication(p1,alice)	publication(p1,bob)
publication(p2,john)	publication(p2,mary)

The hash table created by the algorithm contains the following mapping from constants to variables: { alice $\rightarrow x$, bob $\rightarrow y$, p1$\rightarrow z$, post_quals $\rightarrow u$, assistant_prof $\rightarrow v$}. Note that there are two literals with relation *inPhase*, the first one created using mode definition `inPhase(+,-)` and the second one created using mode definition `inPhase(+,#)`.

3.2 Generalization

After building the bottom-clause associated with a given positive example, the algorithm generalizes the clause to cover more positive examples. It uses the *asymmetric relative minimal generalization* (*armg*) operator to generalize clauses [13]. It performs a beam search to select the best clause generated after multiple applications of the *armg* operator. More formally, given clause C, it randomly picks a subset E^+_S of positive examples to generalize C. For each example $e' \in E^+_S$, it uses the *armg* operator to generate a candidate clause C', which is more general than 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 score of clause is usually computed as the difference between the number of positive and negative examples it covers.

We now explain the *armg* operator in detail. Let C be the bottom-clause associated with example e, relative to I. 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 $C\theta = (T \leftarrow L_1, \cdots, L_i)\theta$ does not cover e', relative to I.Given the bottom-clause C and a positive example e', *armg* drops all blocking atoms from the body of C 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. Because *armg* drops literals from the clause, it is guaranteed that the size of the clause reduces when doing generalization. The bottom-clause is the most specific hypothesis that belongs to the hypothesis space. Therefore, any generalization of it is also in the hypothesis space.

4 SETTING LANGUAGE BIAS

4.1 Generating Predicate Definitions

Let R and S be two relation symbols in the schema of the underlying database. Let $R(e_1, \cdots, e_n)$ and $S(o_1, \cdots, o_m)$ be two atoms in a clause C. Let e_i be the term in attribute $R[A]$ and o_j be the term in attribute $S[B]$, and let e_i and o_j be assigned the same variable or constant. That is, clause C joins R and S on A and B. Clause C is satisfiable only if these attributes share some values in the input database. Typically, the more frequently used joins are the ones over the attributes that participate in inclusion dependencies (INDs), such as foreign-key to primary-key referential constraints. AutoBias uses INDs in the input database to find which attributes, among all relations, share the same type. Let X and Y be sets of attribute names in R and S, respectively. Let I_R and I_S be the relations of R and S in the database. Relations I_R and I_S satisfy *exact IND* (*IND* for short) $R[X] \subseteq S[Y]$ if $\pi_X(I_R) \subseteq \pi_Y(I_S)$. If X and Y each contain only a single attribute, the IND is a *unary IND*. Given IND $R[X] \subseteq S[Y]$ in a database, the database satisfies unary IND $R[A] \subseteq S[B]$, where $A \in X$ and $B \in Y$. INDs are normally stored in the schema of the database. If they are not available in the schema, one can extract them from the database content. We use Binder [42] to discover INDs from the data and produces all unary INDs implied by them. Binder efficiently discovers INDs by using a divide-and-conquer approach. First, it produces all unary candidate INDs. Second, it partitions the input data into small buckets that fit in main memory. Third, it loads each bucket into memory and validates the candidate INDs against the current bucket. It returns all INDs that pass all checks.

We have observed that in some cases using exact INDs is not enough for generating helpful predicate definitions. Consider two attributes A_1 and A_2, which contain values for domains D_1 and D_2, respectively. There may be another attribute A_3 that contains some values from D_1 and some values from D_2. It makes sense to join attributes A_1 (or A_2) with A_3, as A_1 and A_3 contain values for domain D_1. However, exact INDs may not hold between A_1 (or A_2) and A_3. An example of this scenario can be seen in the UW database, whose schema fragments are shown in Table 1. Consider the task of learning a definition for the relation *advisedBy(stud, prof)*, which indicates that the student *stud* is advised by professor *prof.* A relational learning algorithm may learn the following Datalog program for the *advisedBy* relation:

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

which indicates that a student is advised by a professor if they have been co-authors of a publication. This definition requires joining relations *publication*, *student*, and *professor* on attributes *publication[author]*, *student[stud]*, and *professor[prof]*. However, the UW database does not satisfy INDs *publication[author]* $\subseteq$ *student[stud]* or *publication[author]* $\subseteq$ *professor[prof]* because *publication[author]* contains both students and professors.

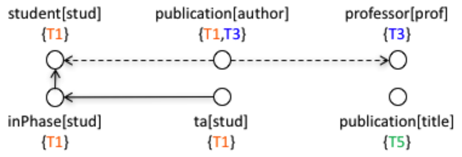

Figure 1: A fragment of the type graph for the UW dataset. Solid lines represent exact INDs and dashed lines represent approximate INDs.

To account for the issue described above, AutoBias also uses *approximate INDs* to assign types to attributes. In an *approximate unary IND* $(R[A] \subseteq S[B], \alpha)$, one has to remove at least α fraction of the distinct values in $R[A]$ so that the database satisfies $R[A] \subseteq S[B]$ [1]. Approximate INDs are not usually maintained in a schema and are instead discovered from the database content. We have implemented a program to extract approximate INDs from the database. We use a relatively high error rate, 50%, for the approximate INDs to allow for a flexible hypothesis space.

After discovering unary exact and approximate INDs, AutoBias runs Algorithm 3 to generate a directed graph called *type graph*, which it then uses to assign types to attributes. First, it creates a graph whose nodes are attributes in the input schema and has an edge between each pair of attributes that participate in an exact or approximate IND. Figure 1 shows an example of the type graph containing a subset of the attributes in the UW schema, where edges corresponding to exact and approximate INDs are shown by solid and dashed lines, respectively. If there are both approximate INDs $(R[A] \subseteq S[B], \alpha_1)$ and $(S[B] \subseteq R[A], \alpha_2)$, AutoBias uses only the one with lower error rate. The algorithm then assigns a new type to every node in the graph without any outgoing edges. For example, it assigns new types T1, T3, and T5 to *student[stud]*, *professor[prof]*, and *publication[title]*, respectively, in Figure 1. If there are cycles in the type graph, the algorithm assigns the same new type to all nodes in each cycle. Next, it propagates the assigned type of each attribute to its neighbors in the reverse direction of edges in the graph until no changes are made to the graph. For example, in Figure 1, the algorithm propagates type T1 to *inPhase[stud]* and *ta[stud]* and attribute *publication[author]* inherits types T1 and T3 from *student[stud]* and *professor[prof]*, respectively. Because the error rates of approximate INDs accumulate over multiple edges in the graph, AutoBias propagates types only once over edges that correspond to approximate INDs.

Given the resulting graph, for each relation, AutoBias computes the Cartesian product of the types associated with its attributes. For each tuple in this Cartesian product, it produces a predicate definition for the relation. For instance, given the type assignment in Figure 1, AutoBias generates predicate definitions `publication(T5,T1)` and `publication(T5,T3)` for the *publication* relation.

Algorithm 3: Algorithm to generate the type graph.

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

4.2 Generating Mode Definitions

AutoBias allows every attribute of each relation be a variable. However, it forces at least one variable in an atom to be an existing variable, i.e., appears in previously added atoms, to avoid generating Cartesian products in the clause. For each attribute A in relation R, AutoBias generates a mode definition for R where attribute A is assigned the + symbol and all other attributes are assigned the − symbol. Hence, all attributes are allowed to have new variables except the attribute with symbol +. For instance, AutoBias generates the mode definitions `publication(+,-)` and `publication(-,+)` for relation *publication* in Table 1.

AutoBias uses a hyper-parameter called *constant-threshold* to determine whether an attribute can be a constant. The value for constant-threshold can take an absolute or a relative threshold. If it is an absolute threshold, AutoBias allows an attribute to be a constant if the number of distinct values in the attribute is below the value of constant-threshold. If it is a relative threshold, AutoBias allows an attribute to be a constant if the ratio of distinct values of the attribute to the total number of tuples in the relation is below the value of constant-threshold. This hyper-parameter must be tuned by the user. As it has a relatively intuitive meaning, it is relatively easy to determine with which values or ranges one should experiment. For each relation R in the database, AutoBias finds all attributes in R that can be constants using the aforementioned rule. Then, it computes the power set $\mathbf{M}$ of these attributes. For each non-empty set $M \in \mathbf{M}$, AutoBias generates a new set of mode definitions where it assigns + and − symbols as described above, except for the attributes in M, which are assigned the # symbol. For example, AutoBias finds that the number of values in attribute *phase* of relation *inPhase* in Table 1 is smaller than the input threshold. Then, this attribute can be constant and AutoBias generates the mode definition `inPhase(+,#)` for relation *inPhase*.

5 SAMPLING HYPOTHESES EFFICIENTLY

Relational learning over large databases is generally time-consuming as the learner has to explore numerous possible hypotheses whose tests of coverage take long. For each bottom-clause, the algorithm has to include literals per tuples that are connected to the positive example and also literals per tuples in the database that are connected to the current ones in the bottom-clause. As numerous tuples across multiple tables may be connected via some join paths to a given positive example over a large database, it may create extremely long bottom-clauses. For example, a bottom-clause may contain tens of thousands of literals over a database with about a million tuples after a couple of iterations. Since the created bottom-clauses contain many literals, it will be time-consuming to generalize the clauses by applying the *armg* operator multiple times. Moreover, the algorithm has to check the number of positive and negative examples covered by the generalized clause in each generalization step. As the clause has many literals, it will be time-consuming to test whether it covers an example.

Experts usually avoid this problem by familiarizing themselves with the underlying domain and database and creating a sufficiently restrictive language bias to limit the hypothesis space and guide the learner. Since AutoBias does not use the experts' intervention and guidance, it may produce language bias that does not sufficiently limit the hypothesis space. Thus, it may take hours if not days for the algorithm to learn a model over a large database.

To address this challenge, in this section, we propose methods to create sufficiently many hypotheses over large data efficiently. As explained in Section 3, AutoBias learning algorithm picks members of its hypothesis space using bottom-clause construction. A bottom-clause C_e associated with an example e is the most specific clause in the hypothesis space that covers e. These bottom-clauses are then generalized to create desired clauses. As described in Section 3, the bottom-clause construction algorithm finds all the information in I relevant to e, denoted by I_e. Then, it creates the bottom-clause C_e by converting tuples in I_e to literals in the bottom-clause. The tuple set I_e may be large if many tuples in I are relevant to e, which in turn makes C_e too large. To overcome this problem, one may use some sampling technique to obtain a smaller tuple set $I_e^s \subseteq I_e$. Ideally, the subset I_e^s contains predictive patterns that will allow the learning algorithm to learn an accurate definition. Then, the algorithm may create a bottom-clause C_e^s with fewer literals than those of C_e from tuples in I_e^s. Clauses C_e and C_e^s have the same head-literal with the information of the underlying example e but the body of C_e^s has fewer literals than that of C_e.

5.1 Naïve Sampling

Let C_e be a bottom-clause associated with example e. A *naïve sample* C_e^s of clause C_e is the clause obtained the following way. Let I_R be the set of tuples in relation R that can be added to I_e^s during the bottom-clause construction. The naïve sampling algorithm obtains a uniform and random sample I_R^s of I_R and adds only the tuples in I_R^s to I_e^s. Let the *inclusion probability* $p(t)$ of tuple $t \in I_e$ be the probability that t is included in I_e^s. In a uniform sample, every tuple in I_R is sampled independently with the same inclusion probability, i.e., $\forall t \in I_R, p(t) = \frac{1}{|I_R|}$. Existing relational learning algorithms use this technique [13, 43].

Nevertheless, in this method, I_e^s may *not* be a random sample of I_e. For example, let this bottom-clause construction algorithm start with example e. Assume that the algorithm has to choose one of tuples t and u in a relation of the underlying database I to add to the bottom-clause. Let t be not connected to any other tuples in I other than e and u be connected to most tuples in I and consequently I_e. Clearly, u is by far more likely to appears in a random sample of I_e than t. Thus, it is reasonable to include u with a higher probability in I_e^s than that of t. But, the naïve sampling method includes t and u with equal probabilities to I_e^s.

As the clauses returned by naïve sampling may not be representative of the underlying hypothesis space, this approach may deliver clauses whose generalizations are not accurate. Moreover, as explained in Section 3, our learning algorithm uses the covering approach (Algorithm 1). Since the generalizations of the returned clauses of this sampling technique may *not* cover sufficiently many positive, each iteration may not lead to removing considerable number of positive examples in the covering approach. Thus, it may also take many iterations and consequently a long time for the learning algorithm to find a reasonably effective model. For instance, consider again the example of the preceding paragraph. Since t is not connected to any other tuples in the database, the bottom-clause may contain only the literal created from t. This clause is too general and may cover most negative examples, therefore, the current iteration of the learning algorithm may not lead to any reasonable clause. Also, this method is biased toward tuples in relations with fewer tuples as it assigns them higher inclusion probabilities.

5.2 Random Sampling

To address the aforementioned shortcomings of the naïve sampling method, one may obtain a random sample of the literals in the body of C_e to construct the body of C_e^s. This method, however, faces two challenges. As explained in Section 3.1, each literal in C_e is head-connected, which means that it is either connected to the head-literal of C_e via some shared variables or it has some variables in common with other literals in the body of C_e that are head-connected. As explained in Section 3.2, a literal that does not meet these conditions, i.e., is not head-connected, will be automatically removed during generalization. Moreover, that literal will not offer any useful information in the database related to the underlying positive example. If one selects literals from the body of C_e uniformly at random, none of the selected literals may not be head-connected. Thus, the learning algorithm may simply return an empty clause after the first step of generalization. Also, if most of the selected literals are not head-connected, the algorithm will eliminate most of the literals in C_e^s after the first step of generalization. Hence, the subsequent generalizations may not have sufficient or interesting information about the underlying example in the database to generalize and learn. One may not get a useful clause that contains predictive information in C_e by simply uniformly and randomly sampling each literal in its body. Thus, every literal in C_e^s must also be head-connected. Moreover, to create C_e^s, one has to obtain a random sample of I_e to construct I_e^s. To create a random sample of I_e, one may construct I_e and then randomly sample sufficiently many of its tuples to construct I_e^s. But, as we explained earlier in this section, I_e may be very time-consuming to construct and materialize for large databases.

To address the aforementioned challenges, we should define a reasonable inclusion probability for each literal in C_e and equivalently each tuple in I_e for the random sample such that the sampled clause does not contain literals that are not head-connected. Furthermore, we should be able to compute these probabilities without computing and materializing C_e and I_e. Next, we precisely compute this inclusion probability without materializing I_e. The *right semi-join* of relations R_1 and R_2 on attributes A and B, denoted as $R_1 \rtimes_{R_1.A=R_2.B} R_2$, is the set of tuples in R_2 such that the values of their attribute B are equal to the value of A of at least one tuple in R_1 [19].

Example 5.1. Consider relations $U_1(A, B)$ and $U_2 = (A, C)$ such that $U_1 = \{(a_1, b_1), (a_2, b_2), \dots, (a_2, b_k)\}$ and $U_2 = \{(a_0, c_1), (a_2, c_2), (a_1, c_3), \cdots, (a_1, c_m)\}$, we have $U_1 \rtimes_{U_1.A=U_2.A} U_2 = \{(a_2, c_2), (a_1, c_3), \cdots, (a_1, c_m)\}$.

For brevity, we call right semi-join simply as semi-join and show $R_1 \rtimes_{R_1.A=R_2.B} R_2$ as $R_1 \rtimes_{A,B} R_2$ unless otherwise noted. The bottom-clause construction algorithm in Section 3.1 is in fact iteratively applying semi-joins to the database relations to add tuples to I_e that are directly or indirectly connected to the positive example e according to the mode and predicate definitions. More precisely, for each pair of attributes of the same type A and B between the target relation T and the relation R in the background knowledge according to the predicate definitions, the bottom-clause construction algorithm will add the tuples of $\{e\} \rtimes R$ to I_e. It then adds the tuples from another relation S to I_e using the semi-join of $\{e\} \rtimes R \rtimes S$. Generally, the algorithm computes $\uplus(\rtimes_{1 \le i \le d-1} R_1 \rtimes \dots \rtimes R_{1+i})$ in its dth iteration where $R_1 = T$ and $\{R_2, \dots, R_d\}$ is a multi-set of possibly non-distinct relations in the background knowledge such that R_i and R_{i+1} have attributes of same type according to the mode definitions. Thus, we should efficiently compute a random sample of every $R_1 \rtimes \dots \rtimes R_{1+i}$.

To compute a random sample of $R_1 \rtimes_{A,B} R_2$, one should materialize $R_1 \rtimes_{A,B} R_2$ and then take a random sample of it. Nonetheless, this defeats the purpose of not computing I_e. Another approach is to take independent random samples of R_1 and R_2 and semi-join them. However, the results may be empty and have very few tuples. Thus, it may take a long time to get a sufficiently large sample. Consider the relations U_1 and U_2 in Example 5.1. It is very unlikely for a random sample of U_1 to contain a tuple whose value for A is a_1 for a sufficiently large k. Also, a random sample of U_2 is unlikely to have a tuple whose A value is a_2 for large values of m. Thus, the semi-join of the random samples of U_1 and U_2 may be empty for a reasonably large number of sampling rounds.

Hence, we extend existing techniques for performing efficient sampling over joins [11, 41, 58] to sample over semi-join $R_1 \rtimes_{A,B} R_2$ efficiently. Let S_1 be such a random sample of R_1. The distributions of attribute values in tuples of S_1 are influenced by the ones of the tuples in R_1. For instance, a random sample of U_1 in Example 5.1 contains mostly tuples whose values of attribute A is a_2. But, the values of attribute A of tuples in $U_1 \rtimes_{A,A} U_2$ are mostly a_1. Thus, one should accept the results of $S_1 \rtimes_{A,B} R_2$ based on the distribution of attribute values in R_2 to create a random sample of $R_1 \rtimes_{A,B} R_2$. Furthermore, let the tuple $t \in R_2$ be the only tuple in R_2 whose value of attribute B is b. Let b appear in the attribute A of only a single tuple of R_1. In this case, t will be the only tuple in $R_1 \rtimes_{A,B} R_2$ whose value of B is b. Now, assume that b appears in the attribute A of more than a single tuple of R_1. This will not change the number of tuples in $R_1 \rtimes_{A,B} R_2$ whose value for attribute B is b. Thus, the distribution of values in $R_1 \rtimes_{A,B} R_2$ depends on the existence of values in $R_1[A]$ but does not change if their frequencies go beyond 1. Therefore, one may randomly select only from values in the set of $\pi_A R_1$ and use it to compute a random sample of the semi-join. Computing the distribution of values of $R_1 \rtimes_{A,B} R_2$ based on the existence of values in $R_1[A]$ instead of their frequencies is the only difference between random sampling over semi-joins as compared to random sampling over joins.

We adapt the extended Olken algorithm [41] for performing random sampling over multi-way joins proposed by Zhao et al. [58] to work over semi-joins. Our sampling algorithm over semi-join $R_1 \rtimes_{R_1.A=R_2.B} R_2$ is as follows. We first select a random value from all values of the set of $\pi_A R_1$ called a. Let $m_{R_2.B}(a)$ denote the frequency of a in attribute B of R_2. Let

$M_{R_2.B}$ is an upper bound on the frequency of each value of B in R_2. From all tuples in R_2 whose values of attribute B is a, we select a tuple t randomly. We accept t with the probability $p = \frac{m_{R_2.B}(a)}{M_{R_2.B}}$ and reject it with $1 - p$. We repeat this process from sampling a value from $\pi_A R_1$ from the beginning until a given number of tuples from R_2 are picked. To compute the values of $m_{R_2.B}(a)$ and $M_{R_2.B}$ and find tuples of R_2 that match a efficiently, we build indexes over the semi-join attributes [11, 41, 58]. To compute the semi-join $R_1 \ltimes R_2 \ldots \ltimes R_n$, we compute the sample S_2 of $R_1 \ltimes R_2$ using the aforementioned algorithm. Then, we compute the sample S_3 of $S_2 \ltimes R_3$ using this algorithm and continue the same process until the sample of semi-join $S_{n-1} \ltimes R_n$ is calculated.

Proposition 5.2. *The aforementioned algorithm produces a random sample* $R_1 \ltimes R_2 \ldots \ltimes R_n$.

Proof. The proof follows the one of random sampling over multi-way joins proposed by Olken [41]. □

The samples of some $S_i \ltimes R_{i+1}$, $1 < i < n$ might be empty as the values in S_i may not match any tuple in R_{i+1}. In this case, one has to repeat the sampling of a preceding binary semi-join to get different values from the ones in S_i. To avoid this problem, we take sufficiently larger number of samples than the desired final number of samples in each binary semi-join.

Given an input number of iterations d, the bottom-clause construction algorithm computes all semi-joins of size up to d and unions their output to construct I_e. To share computation between different samplings, we organize all relations that will be semi-joined according to the predicate definitions in a *semi-join tree* G of depth d. Each node in G is a relation symbol in the schema. The root of G represents the target relation symbol, T. Let n_R be a node in G that represents relation R. A node n_{R_1} in G has a child n_{R_2} if R_1 and R_2 can be semi-joined according to the mode definitions. If the semi-join of R_1 and R_2 is $R_1 \ltimes_{A,B} R_2$, we place the label (A, B) on the edge from n_{R_1} to n_{R_2} in G. Since relation R_2 may appear at the right hand side of multiple semi-joins according to the mode definitions, R_2 may be represented by multiple distinct nodes in G.

Next, we apply the sampling algorithm following edges in G starting from its root to generate the sample of I_e, I_e^s. We consider the example e as the only tuple of the relation of the root of G, which is sampled with probability of 1. This enables us to share and reuse the random sample of a semi-join for the subsequent and longer ones. After sampling the semi-join between a parent n_{R_1} and one of its children n_{R_2}, we add the sampled tuples to I_e^s. We also use this set for the semi-join of n_{R_2} and its children. After constructing I_e^s, we create the bottom-clause C_e^s according to I_e^s. Different paths in G may share some tuples. In this case, the union of randomly sampling from a set of relations is not exactly equivalent to random sampling over the union of the relations [41]. We, however, make the simplifying assumption that they are equivalent to ensure sampling is efficient over large databases. Otherwise, sampling will require considering the intersection of various semi-joins in G, which needs significantly more computations.

5.3 Stratified Sampling

As explained in Section 1, relational learning methods are sometimes used to extract and feed relational features, i.e., features from relations other than the one of the examples, to train non-relational models [24, 31]. In these settings, researchers have found out that using attributes and features from highly connected tuples may *not* be useful as they do not provide enough discriminating information about the target concept or label [24]. They suggest correcting for this bias to deliver a set of diverse attributes or features. Translated to our setting, one may argue that our proposed random sampling algorithm may be biased toward relations and tuples that are strongly connected to other relations and tuples in the database. Thus, it may miss some patterns in the data that effectively define the training examples but are not sufficiently well-connected in the database [24]. To investigate this phenomena, we propose a method that samples a diverse subset of tuples and relationships in the data to construct a sufficiently diverse sample I_e^s of $I_e s$ according to the mode and predicate definitions. Our method provides a sample that contains each possible variation of every literal and ensures that the sampled bottom-clause covers all join paths that connect them according to the language bias.

Let G be a semi-join tree defined in Section 5.2 whose only tuple of its root node is example e. Let S be a relation that contains attribute A, where A can appear as a constant according to the language bias and let n_S is a node that represent S in G. We replace each n_S with a set of new nodes each of which represent a relation that is a subset of S with a distinct value for $S[A]$. The parents of these nodes are the same as n_S. Given a node n_R in G, we define a *stratum* for each child of n_R. Therefore, there is a stratum for each relation S that can join with R and, if S contains an attribute A that can be a constant, there is a stratum for each distinct value in $S[A]$. A *stratified sample* I_e^s of I_e is a subset of I_e that contains at least one tuple for each stratum in G. A stratified sample C_e^s of clause C_e is the clause created from the stratified sample I_e^s of I_e.

Algorithm 4 depicts the bottom-clause construction algorithm using stratified sampling. The algorithm traverses the semi-join tree G in a depth-first manner. Once it reaches a given depth d, it computes the strata in the current relation, e.g., relation S. If S contains an attribute A that can be constant according to the language bias, the algorithm creates a stratum for each distinct value for $S[A]$. If S does *not* contain attributes that can be constant according to the language bias, the only stratum is the set of all tuples in S. It then uniformly samples s tuples for each stratum in S and adds them to I_e^s. Thus, I_e^s is the union of the all sampled strata in S. When the algorithm backtracks to the parent relation R of S, it adds all tuples in R that join the sampled tuples in S to I_e^s.

Since stratified sampling algorithm has to traverse and backtrack nodes in G and perform corresponding operations, it may take longer than random sampling over a large database with a complex schema or language bias specifications. However, it does not need the precomputed statistics and indexes needed to perform random sampling efficiently. It also may not face the problem of empty sampled relations over long semi-joins.

Algorithm 4: Bottom-clause construction algorithm using stratified sampling.

```
Input  : example $e$, # of iterations $d$, sample size $s$
Output : bottom-clause $C_e$
1  $I_e^s = \{\}$
2  foreach attribute $A$ in $e$ do
3      foreach relation $R$ containing attribute $A$ do
4          $I_e^s = I_e^s \cup StratRec(R, A, \{e[A]\}, 1, d, s)$
5  $C_e^s$ = create clause from $e$ and $I_e^s$
6  return $C_e^s$
7  Function StratRec($R, A, M, i, d, s$):
8      $I_e^s = \{\}$
9      $I_R = \sigma_{A \in M}(R)$
10     if $i = d$ (last iteration) then
11         $I_e^s = I_e^s \cup SampleStrata(I_R, s)$
12     else
13         foreach attribute $B$ in $R$ do
14             foreach relation $S$ containing attribute $B$
                do
15                 $I_S = StratRec(S, B, \pi_B(I_R), i+1, d, s)$
16                 $I_e^s = I_e^s \cup (\sigma_{B \in \pi_B(I_S)}(I_R))$
17     return $I_e^s$
```

6 SAMPLING FOR COVERAGE TESTING

As explained in Section 3.2, during generalization, we have to compute the numbers of positive and negative examples covered by a generalized clause to evaluate its quality. One approach is to translate the clause to a Select-Project-Join SQL query and execute it over the underlying data. Nonetheless, these clauses may contain hundreds of literals in the several rounds of generalizations. Our empirical investigations show that it may take a long time to evaluate such SQL queries over a large database. Thus, we follow the approach used in relational learning algorithms and use θ-subsumption to compute the coverage of candidate clauses [36, 38, 43].

In this approach, one builds a *ground bottom-clause* for each positive and negative example using the bottom-clause construction algorithm in Section 3.1 in which constants are *not* replaced with variables. A substitution θ replaces constants and variables in clause C_1 with a set of fresh constants or variables. The resulting clause is denoted as $C_1\theta$. Clause C $theta$-subsumes ground bottom-clause G if and only if there is some substitution $theta$ such that $C\theta \subseteq G$, i.e., the set of literals in the body of $C\theta$ is a subset or equal to the set of literals in the body of G. To test whether a clause covers an example, we check if the clause subsumes the ground bottom-clause of the example. As subsumption testing is NP-hard, we use approximation algorithm to test subsumption of long clauses [30, 36].

Ideally, a ground bottom-clause G_e for example e must contain one literal per each tuple in the database that is connected to e through some joins. Otherwise, the θ-subsumption test may declare that C does *not* cover e when C actually covers e. However, it may be time-consuming to check θ-subsumption for clauses with many literals. Since a learning algorithm performs numerous coverage testing during learning, it is essential to improve the time of coverage testing otherwise learning may take an extremely long time. Hence, we use the three aforementioned sampling techniques to generate ground bottom-clauses. Given that the bottom-clause is built using sampling technique **S**, we also use **S** to generate all ground bottom-clauses for learning.

7 EMPIRICAL STUDY

Data. We run experiments over three datasets. The UW data is explained in Section 1 over which we learn the target relation *advisedBy(stud, prof)*. This dataset contains 9 relations, 1.8K tuples, 102 positive and 204 negative examples. The HIV data contains structural information about chemical compounds (*wiki.nci.nih.gov/display/NCIDTPdata*). We learn the target relation *antiHIV(comp)*, which indicates that compound with id *comp* has anti-HIV activity. This dataset has 5 relations, 7.9M tuples, 2K positive and 4K negative examples. The IMDb data (*imdb.com*) contains information about movies and people who make them. We learn *dramaDirector(dir)*, which indicates that person with id *dir* has directed a drama movie. This database contains 46 relations, 8.4M tuples, 1.8K positive and 3.6K negative examples.

Measure. We compare the quality of the learned definitions using *precision* (*Prec.*) and *recall* [13]. Let the set of true positives for a Horn definition be the set of positive examples in the testing data that are covered by the Horn definition. The precision of a Horn definition is the proportion of its true positives over all examples covered by the Horn definition. The recall of a Horn 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. F-measure (*FM*) is the weighted harmonic mean of the precision and recall. We perform 10-fold cross validation for HIV and IMDb datasets and 5-fold cross validation for UW due to its small size. We evaluate precision, recall, and learning time, showing the average over the cross validation.

Systems. We implement AutoBias over *Castor*, an open source relational learning algorithm that is shown to be more effective that other available systems [43]. It is built on top of VoltDB, *(voltdb.com)*, a main-memory RDBMS. We compare AutoBias against Castor and Aleph [48] as follows. *Castor* assigns the same types to all attributes and allows every attribute to be a variable or a constant. *Castor without constants* (*No const.*) is the same as the baseline method, except that it does not allow any attribute to be a constant. *Castor-Manual tuning* (*Manual*) uses the language bias written by an expert who has knowledge of the relational learning system and knows how to write predicate and mode definitions. The expert had to learn the schema and go through several trial and error phases by running the underlying learning system and observing its results to write the predicate and mode definitions. The expert wrote 19, 14, and 112 predicate and mode definitions for the UW, HIV, and IMDb databases, respectively. *Aleph* is a popular and public relational learning system, which as opposed to Castor does not use relational database systems to store and query the background knowledge and training data. Similar to Auto-Bias, Aleph follows the sequential covering algorithm shown in Algorithm 1. However, Aleph follows a top-down approach. Aleph can emulate multiple relational learning algorithms. We configure Aleph to emulate FOIL [44, 57], which is a popular and well-known top-down relational learning algorithm. QuickFOIL is another implementation of FOIL that uses a relational database system to improve the running time of FOIL [57]. We, however, are not able to find a publicly available version of QuickFOIL. As any other relational learning algorithm, Aleph requires manual tuning to setup its language biases. We use the same predicate and mode definitions used for Castor-Manual tuning. *AutoBias* generates predicate and mode definitions as described in Section 4. The original databases do not contain INDs. AutoBias calls the IND discovery tools explained in Section 4. The preprocessing step to extract INDs takes 1.2 seconds, 1.4 minutes, and 7.8 minutes over the UW, HIV, and IMDb, respectively.

Parameters. We set the constant-threshold hyper-parameter (Section 4.2) to 5 for UW, 80 for HIV, and 400 for IMDb due to their different sizes. Over all settings of Castor and AutoBias, we build bottom-clauses and ground bottom-clauses using naïve sampling to make our results comparable to the ones of Castor. Aleph also uses naïve sampling. We set the sampling rate to at most ten tuples per mode for each dataset. In Section 7.2, we evaluate different sampling techniques. We run experiments on a 2.3GHz Intel Xeon E5-2670 processor, running CentOS Linux 7.2 with 500GB of main memory.

7.1 Approaches to Setting Language Bias

Table 4 illustrates the results of our experiments.

Castor. Over the UW database, Castor is less accurate and efficient compared to other settings. Over the HIV database, Castor obtains competitive precision and recall, but is significantly less efficient than manual tuning and AutoBias. Over the IMDb database, Castor is killed by the kernel because of extreme use of resources. By allowing every attribute to be a constant, every value in the database – even if it has a non-predictive value – may appear in a literal as a constant. Hence, the generated bottom-clause contains too many literals, most of which are not useful for learning a definition. For instance, the first bottom-clause created when running over the IMDb contains on average 1255 literals. By assigning the same type to all attributes, it allows all relations to join with each other on any attribute, resulting in a long running time.

No const. Over UW data, this setting is the most efficient and obtains competitive precision and recall compared to manual tuning and AutoBias. Over the HIV data, it does not terminate after 10 hours. Because no constants are allowed, the system is not able to find any definition that covers many positive examples. Thus, it generates a bottom-clause for each positive example and tries to generalize it to cover more positive examples, which take long. Over the IMDb data, the perfect definition for the target relation *dramaDirector* contains a constant. Hence, this method learns other definitions which are significantly less accurate compared to manual tuning or AutoBias.

Manual. Over the UW and IMDb, Castor with manual tuning results in the most effective definitions. Over the HIV data, it obtains less effective results compared to the baseline or AutoBias. Castor with manual is efficient over all datasets. However, an expert had to spend significant amount of time tuning the language bias. Further, a non-expert user would not be able to specify this bias.

Aleph. Since the top-down learning algorithm used by Aleph is generally biased toward learning relatively short clauses, it is faster than other methods over UW and HIV data. It takes Aleph longer than Castor with manual tuning (Manual) and AutoBias to learn over IMDb, which is because it does not use any underlying database system to access and query data. This approach does not scale for databases with numerous tuples, such as IMDb. Aleph with manual tuning delivers less effective definitions than those by Castor with manual tuning and AutoBias over all datasets.

AutoBias. In general, AutoBias is more effective than Castor, No const, Aleph, and almost as effective as manual tuning. AutoBias is less efficient than manual tuning. Manually written predicate and mode definitions provide a more restricted hypothesis space than the ones generated by AutoBias. Thus, it has to explore a larger hypothesis space. Nevertheless, the overhead in the running time is about 13 minutes for the HIV data and 4 minutes for the IMDb data, which is a reasonable overhead for saving an expert's time and the enterprise's financial resources that pay the machine learning expert. There is no overhead over the UW database. Hence, we argue that automating the generation of predicate and mode definitions with the cost of a modest overhead in performance is a reasonable trade-off. Further, AutoBias enables non-experts to use learning systems easily.

7.2 Sampling Techniques

We have implemented three versions of AutoBias by modifying the modules in charge of generating the bottom-clause construction and coverage testing. Each version uses a different sampling technique for bottom-clause construction: **Naïve** uses naïve sampling, **Random** uses random sampling, and **Stratified** for stratified sampling. Each sampling method is used for both bottom-clause construction and coverage testing. We use the sampling rate of at most 10 tuples per each mode for all sampling methods and datasets. We have run random and stratified sampling methods over each dataset five times and computed the average of the resulting runs.

Table 5 shows the effectiveness and efficiency of learning using the aforementioned sampling techniques over UW, IMDb, and HIV. Random delivers higher efficiency than other methods over large datasets of IMDb and HIV, which confirms that Random selects more promising bottom-clauses that in turn constructs an accurate model fast and in relatively few iterations. This difference is more significant over HIV data. This dataset is large and has a relatively complex schema with significant diversities in values and relationships. Moreover, its target relation is complex and there is not any definition with a reasonably small literals and clauses that covers all positive examples and does not cover any negative ones. For example, each compound in this data may contain hundreds of atoms. Some atoms are common elements, e.g., Hydrogen, while other atoms are rare elements, e.g., Lithium. Random is able to explore join paths that lead to all types of elements in a compound. The bottom-clauses generated by Random contain diverse information, which allows it to learn more accurate definitions.

All methods return the (same) effective definition for the IMDb database. The accurate definition over this dataset is relatively short. Our observations indicate that when there is a relatively short definition that exactly represents the training data all methods can find an effective definition for the target relation. Random, however, finds this definition faster than other methods due to the reasons explained in the preceding paragraph.

Table 4: Results of learning relations over UW, IMDb, and HIV (h=hours, m=minutes, s=seconds).

Data	Measure	Castor	No const.	Manual	Aleph	AutoBias
UW	Prec.	0.76	0.96	0.93	0.78	0.84
	Recall	0.50	0.48	0.54	0.17	0.54
	FM	0.60	0.64	0.68	0.27	0.64
	Time	47s	6.6s	11s	3.5s	24.4s
IMDb	Prec.	-	0.68	1	0.66	1
	Recall	-	0.51	0.99	0.44	0.99
	FM	-	0.58	0.99	0.52	0.99
	Time	-	9.2h	2.7m	6.4m	3.21m
HIV	Prec.	0.80	-	0.74	0.72	0.80
	Recall	0.83	-	0.84	0.69	0.85
	FM	0.81	-	0.78	0.70	0.82
	Time	59.7m	>10h	22.6m	6.2m	35.1m

Table 5: Results over UW, HIV, and IMDb data with different sampling techniques. (m=minutes, s=seconds).

Data	Measure	Naïve	Random	Stratified
UW	FM	0.64	0.61	0.54
	Time	24.4s	50.23s	37.86s
IMDb	FM	0.99	0.99	0.99
	Time	3.21m	3.13m	4.05m
HIV	FM	0.82	0.83	0.79
	Time	35.1m	21.87m	34.16m

Due to the small size of the data and schema of UW, Naïve is able to create a sufficiently representative sample of the data and learn an effective definition over this dataset. This indicate that Random sampling technique, similar to other sampling methods, may not be useful over small datasets and shows its advantages over sufficiently large databases. In particular, due to its overhead to select a random sample of the underlying hypothesis space, it may take longer than Naïve over small datasets. Stratified performs less effective and efficient than other approaches, which indicate that the observations made for non-relational models using relational features may not hold in relational learning setting. It indicates that a random sample over the hypothesis space provides a sufficiently representative set of hypotheses. It is due to a different type of hypothesis space in non-relational and relational models.

8 RELATED WORK

Recently, there has been a growing interest in relational learning algorithms that scale to large data in both the database and machine learning communities [18, 33, 43, 56, 57]. Researchers have used differentiable matrix operations to learn relational models over RDF data [54, 55]. These methods are limited to datasets with relatively small number of distinct values, e.g., tens of thousands, and short clauses, e.g., at most 3 body literals [55].

REFERENCES

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

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

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

[4] Marcelo Arenas, Gonzalo I. Diaz, and Egor V. Kostylev. 2016. Reverse Engineering SPARQL Queries. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16).* International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 239–249. https://doi.org/10.1145/2872427.2882989

[5] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Ben Zorn. 2014. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. In *PLDI '15 Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pldi 2015 ed.). Microsoft Research Technical Report. Distinguished Artifact Award.

[6] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv* (2018). https://arxiv.org/pdf/1806.01261.pdf

[7] Michael Benedikt, Kristian Kersting, Phokion G. Kolaitis, and Daniel Neider. 2019. Logic and Learning (Dagstuhl Seminar 19361). *Dagstuhl Reports* 9, 9 (2019), 1–22. https://doi.org/10.4230/DagRep.9.9.1

[8] Angela Bonifati, Radu Ciucanu, and Sławek Staworko. 2016. Learning Join Queries from User Examples. *ACM Trans. Database Syst.* 40, 4, Article 24 (Jan. 2016), 38 pages. https://doi.org/10.1145/2818637

[9] Angela Bonifati, Ugo Comignani, Emmanuel Coquery, and Romuald Thion. 2019. Interactive Mapping Specification with Exemplar Tuples. *ACM Trans. Database Syst.* 44, 3, Article 10 (June 2019), 44 pages. https://doi.org/10.1145/3321485

[10] Peter Buneman and Wang-Chiew Tan. 2019. Data Provenance: What Next? *SIGMOD Rec.* 47, 3 (Feb. 2019), 5–16. https://doi.org/10.1145/3316416.3316418

[11] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1999. On Random Sampling over Joins. In *SIGMOD Conference.*

[12] William W. Cohen, Haitian Sun, R. Alex Hofer, and Matthew Siegler. 2020. Scalable Neural Methods for Reasoning With a Symbolic Knowledge Base. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020.* OpenReview.net. https://openreview.net/forum?id=BJlguT4YPr

[13] Luc De Raedt. 2010. *Logical and Relational Learning* (1st ed.). Springer Publishing Company, Incorporated.

[14] Daniel Deutch and Amir Gilad. 2019. Reverse-Engineering Conjunctive Queries from Provenance Examples. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 277–288. https://doi.org/10.5441/002/edbt.2019.25

[15] Pedro Domingos. 2018. Machine Learning for Data Management: Problems and Solutions. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18).* Association for Computing Machinery, New York, NY, USA, 629. https://doi.org/10.1145/3183713.3199515

[16] Richard Evans and Edward Grefenstette. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Intell. Res.* 61 (2018), 1–64.

[17] Anna Fariha and Alexandra Meliou. 2019. Example-Driven Query Intent Discovery: Abductive Reasoning Using Semantic Similarity. *Proc. VLDB Endow.* 12, 11 (July 2019), 1262–1275. https://doi.org/10.14778/3342263.3342266

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

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

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

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

[22] Sumit Gulwani. 2017. Research for Practice: Programming by Examples. *Research for Practice, CACM* 60 (July 2017), 46–49. https://www.microsoft.com/en-us/research/publication/research-practice-programming-examples/

[23] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB* 5, 12 (2012).

[24] David D. Jensen and Jennifer Neville. 2002. Linkage and Autocorrelation Cause Feature Selection Bias in Relational Learning. In *Machine Learning, Proceedings of the Nineteenth International Conference (ICML 2002), University of New South Wales, Sydney, Australia, July 8-12, 2002.* 259–266.

[25] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 683–698. https://doi.org/10.1145/3035918.3064034

[26] Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18).* Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/3183713.3183727

[27] Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18).* ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/3183713.3183727

[28] Angelika Kimmig, David Poole, and Jay Pujara. 2020. Statistical Relational AI (StarAI) WorkShop. In *AAAI.*

[29] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD.*

[30] Ondrej Kuzelka and Filip Zelezný. 2008. A Restarted Strategy for Efficient Subsumption Testing. *Fundam. Inform.* 89 (2008), 95–109.

[31] Ni Lao, Einat Minkov, and William Cohen. 2015. Learning Relational Features with Backward Random Walks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).* Association for Computational Linguistics, Beijing, China, 666–675. https://doi.org/10.3115/v1/P15-1065

[32] Hao Li, Chee Yong Chan, and David Maier. 2015. Query From Examples: An Iterative, Data-Driven Approach to Query Construction. *PVLDB* 8 (2015), 2158–2169.

[33] Marcin Malec, Tushar Khot, James Nagy, Erik Blasch, and Sriraam Natarajan. 2016. Inductive logic programming meets relational

databases: An application to statistical relational learning. In *ILP*.

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

[35] Tom Mitchell. 1997. *Machine Learning*. McGraw-Hil.

[36] Stephen Muggleton. 1995. Inverse entailment and Progol. *New Generation Computing* 13 (1995), 245–286.

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

[38] Stephen Muggleton, Jose Santos, and Alireza Tamaddoni-Nezhad. 2009. ProGolem: A System Based on Relative Minimal Generalisation. In *ILP*.

[39] Arvind Neelakantan, Quoc V. Le, Martín Abadi, Andrew McCallum, and Dario Amodei. 2017. Learning a Natural Language Interface with Neural Programmer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ry2YOrcge

[40] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. 2016. Neural Programmer: Inducing Latent Programs with Gradient Descent. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.04834

[41] Frank Olken. 1993. *Random Sampling from Databases*. Ph.D. Dissertation. UC Berkeley.

[42] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & Conquer-based Inclusion Dependency Discovery. *PVLDB* 8 (2015), 774–785.

[43] Jose Picado, Arash Termehchy, and Alan Fern. 2017. Schema Independent Relational Learning. In *SIGMOD Conference*.

[44] J. Ross Quinlan. 1990. Learning Logical Definitions from Relations. *Machine Learning* 5 (1990), 239–266.

[45] Luc De Raedt, David Poole, Kristian Kersting, and Sriraam Natarajan. 2017. Statistical Relational Artificial Intelligence: Logic, Probability and Computation. In *NeurIPS*.

[46] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

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

[48] Ashwin Srinivasan. 2004. *The Aleph Manual*.

[49] Wei Chit Tan, Meihui Zhang, Hazem Elmeleegy, and Divesh Srivastava. 2017. Reverse Engineering Aggregation Queries. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1394–1405. https://doi.org/10.14778/3137628.3137648

[50] Balder ten Cate, Víctor Dalmau, and Phokion G. Kolaitis. 2013. Learning schema mappings. *ACM Trans. Database Syst.* 38, 4 (2013), 28:1–28:31. https://doi.org/10.1145/2539032.2539035

[51] Balder ten Cate, Phokion G. Kolaitis, Kun Qian, and Wang-Chiew Tan. 2017. Approximation Algorithms for Schema-Mapping Discovery from Data Examples. *ACM Trans. Database Syst.* 42, 2 (2017), 12:1–12:41. https://doi.org/10.1145/3044712

[52] Balder ten Cate, Phokion G. Kolaitis, Kun Qian, and Wang-Chiew Tan. 2018. Active Learning of GAV Schema Mappings. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, 355–368. https://doi.org/10.1145/3196959.3196974

[53] William Yang Wang and William W. Cohen. 2015. Joint Information Extraction and Reasoning: A Scalable Statistical Relational Learning Approach. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics, 355–364. https://doi.org/10.3115/v1/p15-1035

[54] William Yang Wang and William W. Cohen. 2016. Learning First-Order Logic Embeddings via Matrix Factorization. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, Subbarao Kambhampati (Ed.). IJCAI/AAAI Press, 2132–2138. http://www.ijcai.org/Abstract/16/304

[55] Fan Yang, Zhilin Yang, and William W. Cohen. 2017. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 2319–2328. http://papers.nips.cc/paper/6826-differentiable-learning-of-logical-rules-for-knowledge-base-reasoning

[56] Xiaoxin Yin, Jiawei Han, Jiong Yang, and Philip S. Yu. 2004. CrossMine: efficient classification across multiple database relations. *ICDE* (2004), 399–410.

[57] Qiang Zeng, Jignesh M. Patel, and David Page. 2014. QuickFOIL: Scalable Inductive Logic Programming. *PVLDB* 8 (2014), 197–208.

[58] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*.