Scalable and Usable Relational Learning With Automatic Language Bias

Jose Picado
Oregon State University
picadolj@oregonstate.edu

Arash Termehchy
Oregon State University
termehca@oregonstate.edu

Alan Fern
Oregon State University
alan.fern@oregonstate.edu

Sudhanshu Pathak
Oregon State University
pathaks@oregonstate.edu

Ilango, Praveen
Oregon State University
ilangop@oregonstate.edu

John Davis
Oregon State University
davisjo5@oregonstate.edu

ABSTRACT

A large body of machine learning and AI is focused on learning models composed of (probabilistic) logical rules, i.e., relational models, over relational databases and knowledge bases. To learn effective relational models over the huge space of possible ones efficiently, users of the current learning systems must restrict the structure of the candidate models using *language bias*. ML experts have to spend a long time inspecting the data and performing many rounds of trial and error to develop an effective language bias. We propose AutoBias, a system that leverages information in the underlying data to generate the language bias. As its induced language bias may not restrict the set of candidate models as tightly as the manually-written ones, learning may not scale to large datasets. Thus, we design novel and efficient methods to sample and learn effective relational models over large data. Our extensive empirical study shows that AutoBias delivers the same accuracy as using manually-written language bias by imposing only a slight overhead on the learning time.

CCS CONCEPTS

• Information systems → Data management systems; **• Computing methodologies → Logical and relational learning**;

KEYWORDS

Scalable Relational Learning; Language Bias; Sampling

ACM Reference Format:
Jose Picado, Arash Termehchy, Alan Fern, Sudhanshu Pathak, Ilango, Praveen, and John Davis. 2021. Scalable and Usable Relational Learning With Automatic Language Bias. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3448016.3457275

1 INTRODUCTION

Relational Learning. 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, 30, 33, 46, 48, 55, 56]. Consider the UW database (*alchemy.cs.washington.edu/data/uw-cse*), which contains information about a computer science department whose schema fragments are shown in Table 2. 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, 30, 33, 44, 46, 56, 58]. Learned definitions are usually (probabilistic) first-order logic formulas and often restricted to Datalog programs. From the set of all possible Datalog programs, the learning algorithm returns the one that covers the most positive and fewest negative examples in the data. For example, given some positive and negative examples of relation *advisedBy(stud,prof)* and other existing relation instances of the schema in Table 2, a relational learning algorithm may learn the following definition for *advisedBy*:

$$\begin{aligned} advisedBy(x,y) \leftarrow\ & student(x), professor(y), \\ & publication(z,x), publication(z,y), \\ & ta(t,x,u), taughtBy(t,y,u) \end{aligned}$$

which means that if a student is a co-author on some publication with a professor and is also a TA of a course taught by the professor, the student is advised by the professor.

Advantages of Relational Learning. Non-relational learning methods, e.g., logistic regression, rely on the assumption that the data points are independent and follow an identical distribution (IID) [36]. The IID assumption is often violated over relational data, therefore, using non-relational models may result in inaccurate models that are too biased to the training data [13, 15, 21, 46]. Relational models are also interpretable and easy to understand. Moreover, as relational models directly leverage the structure of the data, users do not need to perform the lengthy process of feature engineering. Interestingly, when users choose to use non-relational models over structured data, they often use relational learning methods to extract relevant features for their models [24, 30]. Thus, relational models are widely used in AI, e.g., information extraction [15, 28, 46, 54], question answering [40, 41, 46, 56]; data management, usable query interfaces [3, 4, 8, 17, 26, 27, 31, 49], entity resolution [7, 16, 20], schema mapping discovery [7, 9, 50–52], data cleaning [5, 25, 47], data provenance [10, 14]; and programming languages and software engineering [22].

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457275

Challenges of Scaling Relational Learning. It is very difficult to scale relational learning to large data for two reasons. First, the space of possible hypotheses that a relational learning algorithm should explore consists of all Datalog programs defined over the schema of the underlying data, which is enormous over a large database [6, 12, 44, 46, 58]. Second, a relational learning algorithm has to evaluate the quality of each hypothesis in the space, i.e., whether it covers sufficiently many positive and few negative examples, to pick the most effective one. It generally takes a long time to evaluate each hypothesis over large data. In particular, the Datalog programs that explain the data often have many literals and complex structures, e.g., joins of many relations.

Current Approach: Manual Language Bias. To address the first aforementioned challenge, users constrain the hypothesis space of the algorithm using a type of inductive bias called *language bias* [13]. Language bias restricts the relations, the join paths between the relations, and the values used in the hypotheses to ensure that the hypothesis space is both sufficiently small and contains promising definitions. To develop accurate language bias, a user should know both the internals of the learning algorithm and the schema and content of the database well. They should also have a clear intuition on the structure of accurate models. However, most users are *not* sufficiently familiar with the structure of the data and accurate models as well as internals of learning algorithms. Due to the huge volume, complex structures, or frequent evolution of datasets, it is challenging for users to gain knowledge about the data and structure of promising definitions for large datasets. Currently, language bias is developed by ML experts via a tedious and lengthy process of trial and error and may consist of hundreds of rules, which are hard to maintain. *Language bias is called the "black magic" needed to make relational learning work* [6].

Our Contributions. In this paper, we propose a novel and usable system called AutoBias that scales relational learning to large data. We proposed novel methods to generate language bias automatically with minimal user intervention. We also propose new techniques to explore the hypothesis space induced by the language bias and evaluate candidate models efficiently over large data. Our contributions are:

- We propose a novel method that leverages the exact and approximate database constraints [1, 2, 43] and content to induce language bias automatically (Section 3). These constraints are usually available in the schema or can be discovered from the database [1, 19, 43].
- 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 a large hypothesis space efficiently (Section 4.2). It is challenging to randomly sample the hypothesis space over relational data as it requires the construction and materialization of all possible Datalog definitions over the data before selecting a random sample, which takes a very long time. We propose an efficient method to construct only randomly chosen definitions.
- Randomly sampled definitions might be biased toward highly connected tuples or relations in the data, which may reduce the accuracy of learned model. Thus, we propose a stratified sampling method that delivers more diverse hypotheses than random sampling (Section 4.3).

Table 1: Relational Learning Notations

$R(e_1, \dots, e_n)$	literal
$T(\bar{x}) \leftarrow R_1(\bar{u_1}) \dots R_n(\bar{u_n})$	clause
$I \wedge C \models e$	clause C covers e given I

Table 2: Schema for the UW data.

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)	

- A relational learning algorithm must evaluate the quality of each potential hypothesis. Since each hypothesis may contain hundreds of literals, 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 5).
- 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 the random sampling approach improves the efficiency of our system significantly and delivers more effective or as effective results than the other techniques over large databases.

2 BACKGROUND

2.1 Basic Definitions

An *atom* is a formula in the form of $R(e_1, \dots, e_n)$, 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 called *terms*.

Definition 2.1. A Horn clause (clause for short) is a finite set of literals that contains exactly one positive literal called head-literal.

Definition 2.2. A Horn definition (definition for short) is a set of clauses with the same head-literal.

A relational learning algorithm learns definitions from input relational databases and training data. The learned definitions are usually restricted to non-recursive Datalog programs without negation for efficiency reasons.

Definition 2.3. The hypothesis space is the set of all definitions that a relational learning algorithm explores. Each member of the hypothesis space is a hypothesis.

Definition 2.4. 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.

Horn definition H covers an example e if at least one of its clauses covers e. Relational learning algorithms search over the hypothesis space for a definition that covers as many positive and few negative examples as possible.

2.2 Language Bias

Language bias is a set of predicate and mode definitions [13].

Table 3: Predicate and mode definitions for UW data.

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(-,+)

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 3, 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 of 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 3 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 can 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 2.3.1.

2.3 Relational Learning Algorithms

AutoBias uses the same learning algorithm as existing relational learning systems [46]. In this section, we explain this algorithm and how it uses language bias. Like other relational learning systems, AutoBias uses a sequential covering approach [33, 38, 44, 46, 54, 58]. Algorithm 1 depicts this approach. 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.

It is shown that the most effective way to implement the *LearnClause* function is with the bottom-up approach, in which the algorithm first finds relevant patterns in the data and then generalizes them to find clauses that explain the training examples accurately [35, 38, 44]. Thus, we use this approach. It has two main steps: a Bottom-clause Construction step and a Generalization step.

Algorithm 1: Sequential covering algorithm.

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

$H = \{\}$
$U = E^+$
while U *is not empty* **do**
  $C = LearnClause(I, U, E^-)$
  if C *satisfies minimum criterion* **then**
    $H = H \cup C$
    $U = U - \{e \in U | H \wedge I \models e\}$
return H

Algorithm 2: Bottom-clause construction.

Input : example e, # of iterations d, sample size s
Output : BC C_e

$I_e = \{\}$
$M = \{\}$ // M stores known constants
add constants in e to M
for $i = 1$ *to* d **do**
  foreach *relation* $R \in I$ **do**
    foreach *attribute* A *in* R **do**
      $I_R = \sigma_{A \in M}(R)$
      foreach *tuple* $t \in I_R$ **do**
        add t to I_e and constants in t to M
C_e = create clause from e and I_e
return C_e

2.3.1 Bottom-clause Construction. A *bottom-clause* (BC for short) 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 BC construction algorithm consists of two phases: 1) it finds all the set of tuples $I_e \subseteq I$ that are connected to e, and 2) given I_e, it creates the BC C_e. Algorithm 2 shows the BC construction algorithm. Learning systems use predicate and mode definitions to restrict the structure and syntax of the BCs [13, 44].

More precisely, 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 considers only attributes that 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 symbol # on attribute R. If an attribute A can be both a variable and a constant, the algorithm creates two new literals, one for each case. 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 a given number of iterations d for efficiency.

Time Complexity. At each iteration of BC construction, the algorithm must find database tuples that contain constants in the literals of the current BC. As the algorithm terminates after a fixed number of iterations, its time complexity is linear in the size of the database. It is challenging to scale BC construction to large databases.

Example 2.5. Consider the database I in Table 4, the predicate and mode definitions in Table 3, and a positive example e, which is *advisedBy(juan,sarita)*. Given that d is 1, the BC associated with e and relative to I is:

$$
\begin{aligned}
&advisedBy(x, y) \leftarrow student(x), professor(y),\\
&\quad inPhase(x, u), inPhase(x, post\_quals), hasPosition(y, v),\\
&\quad publication(z, x), publication(z, y).
\end{aligned}
$$

The hash table created by the algorithm contains the following mapping from constants to variables: {juan $\rightarrow x$, sarita $\rightarrow y$, 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(+,#)`.

Table 4: Fragments of the UW database.

student(juan)	professor(sarita)
student(john)	professor(mary)
inPhase(juan,post_quals)	hasPosition(sarita,assistant_prof)
inPhase(john,post_quals)	hasPosition(mary,associate_prof)
publication(p1,juan)	publication(p1,sarita)
publication(p2,john)	publication(p2,mary)

2.3.2 Generalization. After building the BC 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 a clause is usually computed as the difference between the number of positive and negative examples it covers.

We now explain the *armg* operator. Let C be the BC 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 BC 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.

Time Complexity. To capture relevant patterns in a large database, a BC usually contains hundreds of literals. Hence, clauses processed during most generalization steps contain hundreds of literals, i.e., hundreds of joins. Using proper indexes, coverage computation of a clause is linear to the size of the database. The generalization algorithm itself is quadratic in the number of literals in the generalized clause. As each clause is significantly smaller than the database, coverage computation dominates the time of generalization.

3 SETTING LANGUAGE BIAS

In this section, we propose methods to generate predicate and mode definitions automatically. We use exact or approximate database constraints to induce predicate definitions and the cardinality attributes to generate mode definitions.

3.1 Generating Predicate Definitions

Let R and S be two relation symbols in the schema of the underlying database. Let $R(e_1, \cdots, e_n)$ and $S(o_1, \cdots, o_m)$ be two atoms in a clause C. Let e_i be the term in attribute $R[A]$ and o_j be the term in attribute $S[B]$, and let e_i and o_j be assigned the same variable or constant. That is, clause C joins R and S on A and B. Clause C is satisfiable only if these attributes share some values in the input database. Typically, the more frequently used joins are the ones over the attributes that participate in inclusion dependencies (INDs), such as foreign-key to primary-key referential constraints. 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 [43] to discover INDs from the data and produce 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 2. 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]* ⊆ *student[stud]* or *publication[author]* ⊆ *professor[prof]* because *publication[author]* contains both students and professors.

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

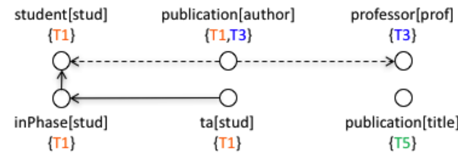

Figure 1: Part of the type graph for the UW data. Solid and dashed lines show exact and approximate INDs.

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 Σ.
Output: Type graph G.

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

3.2 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 2.

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 2 is smaller than the input threshold. Then, this attribute can be constant and AutoBias generates the mode definition `inPhase(+,#)` for relation *inPhase*.

4 EFFICIENT BC CONSTRUCTION

The BC construction algorithm finds all the information in I relevant to example e, denoted by I_e. Then, it creates the BC C_e associated with e by converting tuples in I_e to literals in the BC. I_e is often large as many tuples in I are usually relevant to e, which in turn makes C_e too large. As the time of creating BCs is linear to the size of the database, it takes a long time to create a BC over a large dataset. It may take significantly longer for AutoBias as the its induced language bias may not restrict the hypothesis space as much as manually tuned ones. To overcome this problem, one may use some *sampling technique* to obtain a smaller tuple set $I_e^s \subseteq I_e$. Then, the algorithm may create a BC C_e^s from tuples in I_e^s that has significantly fewer literals than C_e. The subset I_e^s should contain representative and predictive patterns that allow the learning algorithm to learn an accurate definition.

4.1 Naïve Sampling

4.1.1 Naïve Sampling Algorithm. A *naïve sample* C_e^s of clause C_e is the clause obtained the following way [13, 44]. Let I_R be the set of tuples in relation R that can be added to I_e^s during BC 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., $\forall t \in I_R, p(t) = \frac{1}{|I_R|}$.

Time Complexity. The naïve sampling partially scans each relation to sample a subset of its tuples, therefore, its time complexity is linear to the size of the underlying data.

4.1.2 Shortcomings of Naïve Sampling. This method is biased toward tuples in relations with fewer tuples. It may add non-relevant literals from the small relations, i.e., small $|I_R|$, and ignore the relevant ones from the large ones. Using Example 2.5, many more literals from *hasPosition* may be added to the BC compared to *publication* due to the size of the relations. The result is an inaccurate definition for *advisedBy*. Moreover, it delivers a BC that does not contain a representative and random sample of relational patterns in the data.

Consider again Example 2.5. In the original UW database, there are many instances of the co-author relationship represented by the self-join of *publication*. But, because the naïve method samples tuples from *publication* relations independently from each other, it may not include any instance of this relationship in its created BC. This leads to learning inaccurate results. Furthermore, since the generalizations of the BCs returned by this method may not cover sufficiently many positive examples, each iteration may not lead to removing a 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.

4.2 Random Sampling

4.2.1 Challenges of Randomly Sampling Over BCs. To address the aforementioned shortcomings of the naïve sampling algorithm, one may obtain a random sample of the literals in the body of C_e to construct a small and representative clause C_e^s. This method, however, faces three following challenges.

First, as explained in Section 2.3.1, 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 2.3.2, a literal that is not head-connected will be removed during generalization as it does not offer any useful information about the underlying positive example. If one selects literals from the body of C_e uniformly at random, most or all the selected literals may not be head-connected. Hence, the subsequent generalizations may not have sufficient information about the underlying example and deliver an inaccurate or empty clause. Thus, every literal in C_e^s must also be head-connected.

Second, a random sample of C_e must reflect a random sample of the relationships between literals in C_e, i.e., the more connected and related a literal is to other literals in C_e, the more likely it is for that literal to be in the randomly sampled clause. Third, it is too time-consuming to construct and materialize C_e over large data. Hence, we have to create C_e^s without materializing C_e.

4.2.2 Using Semi-Joins to Compute Inclusion Probability. 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 and reflect the relationships between literals in C_e. Furthermore, we should be able to compute these probabilities without 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 4.1. Consider relations $U_1(A, B)$ and $U_2 = (A, C)$ such that $U_1 = \{(a_1, b_1), (a_2, b_2), \ldots, (a_2, b_k)\}$ 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 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 BC construction algorithm in Section 2.3.1 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 BC 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 $\uplus(\rtimes_{1 \leq i \leq d-1} R_1 \rtimes \ldots \rtimes R_{1+i})$ in its dth iteration where $R_1 = T$ and $\{R_2, \ldots, R_d\}$ 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 $R_1 \rtimes \ldots \rtimes R_{1+i}$.

4.2.3 Random Sampling Over Semi-Joins. 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 or 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 4.1. 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.

Hence, we extend existing techniques for performing efficient sampling over joins [11, 42, 59] 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 4.1 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 and over joins.

We adapt the extended Olken algorithm [42] for performing random sampling over multi-way joins proposed by Zhao et al. [59] 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 $M_{R_2.B}$ be 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, 42, 59]. To compute the semi-join $R_1 \rtimes R_2 \ldots \rtimes R_n$, we compute the sample S_2 of $R_1 \rtimes R_2$ using the aforementioned algorithm. Then, we compute the sample S_3 of $S_2 \rtimes R_3$ using this algorithm and continue the same process until the sample of semi-join $S_{n-1} \rtimes R_n$ is calculated. The proof of the next proposition follows the one of random sampling over multi-way joins in [42].

PROPOSITION 4.2. *The aforementioned algorithm produces a random sample $R_1 \rtimes R_2 \ldots \rtimes R_n$.*

The samples of some $S_i \rtimes R_{i+1}$, $1 < i < n$ 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.

4.2.4 Random Sampling Over BC. Given an input number of iterations d, the BC 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 $R_1 \rtimes_{A,B} R_2$, 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 BC 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 [42]. We, however, make the simplifying assumption that they are equivalent to ensure sampling is efficient over large databases. Otherwise, sampling requires considering the intersection of various semi-joins in G that needs significantly more computations.

Time Complexity. Using indexes, the time complexity of the random sampling is linear to the size of the underlying database. But, similar to naïve sampling, random sampling checks only a small sample of tuples in the underlying data, therefore, it is significantly faster creating a BC using the full information of each relation. Additionally, it delivers a more representative BC than the one returned by naïve sampling, which leads to finding an effective definition sooner in the generalization step as shown in our empirical studies.

4.3 Stratified Sampling

4.3.1 Issues of Random Sampling. As explained in Section 1, relational learning methods are sometimes used to extract and feed relational features to train non-relational models [24, 30]. 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 [24]. 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 patterns that effectively predict the target concept but are not sufficiently well-connected.

For instance, in the original UW database whose fragments are used in Example 2.5, the TAship relationship between a student and professor, represented by the join of *ta* and *taughtBy*, may be an effective feature to indicate that the student is being advised by the professor. The number of instances of TAship relationship is significantly less than that of the co-authorship relationship. It is likely that the random sampling algorithm does *not* include an instance of TAship relationship in its BC and misses this feature.

Algorithm 4: Stratified Sampling Algorithm.

Input : example e, # of iterations d, sample size s
Output: BC C_e

$I_e^s = \{\}$
foreach *attribute A in e* **do**
  foreach *relation R containing attribute A* **do**
    $I_e^s = I_e^s \cup StratRec(R, A, \{e[A]\}, 1, d, s)$
C_e^s = create clause from e and I_e^s
return C_e^s
Function `StratRec`(R, A, M, i, d, s):
  $I_e^s = \{\}$
  $I_R = \sigma_{A \in M}(R)$
  if $i = d$ *(last iteration)* **then**
    $I_e^s = I_e^s \cup SampleStrata(I_R, s)$
  else
    foreach *attribute B in R* **do**
      foreach *relation S containing attribute B* **do**
        $I_S = StratRec(S, B, \pi_B(I_R), i + 1, d, s)$
        $I_e^s = I_e^s \cup (\sigma_{B \in \pi_B(I_S)}(I_R))$
  return I_e^s

4.3.2 Stratified Sampling of a BC. 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 BC covers all join paths that connect them according to the language bias. Let G be a semi-join tree defined in Section 4.2 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 be a node that represents 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 BC 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.

Time Complexity. The stratified sampling algorithm traverses and backtracks nodes in G and performs corresponding operations. Thus, its time complexity is linear in the number of tuples in the database. As it inspects all tuples in the examined relations to construct strata, it takes longer than naïve and random sampling. As opposed random sampling, it does not need the precomputed statistics and indexes to perform sampling efficiently.

5 EFFICIENT COVERAGE TESTING

Coverage Testing As Query Execution. As explained in Section 2.3.2, the most time-consuming step in generalization is evaluating the quality of each generalized clause by computing the number of positive and negative examples covered by the clause. One approach is to translate the clause to a Select-Project-Join SQL query and execute it over the underlying data. These clauses may contain hundreds of literals that translates to queries with hundreds of joins. It is very time-consuming to run such queries over large data.

Using θ-Subsumption. Thus, we use θ-subsumption to compute the coverage of clauses [37, 39, 44]. In this approach, one builds a *ground BC* for each positive and negative example using the BC construction algorithm in Section 2.3.1 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 BC 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 BC of the example. As subsumption testing is NP-hard, we use an approximation algorithm to test subsumption [29, 37].

Efficient θ-Subsumption Using Sampling. Ideally, a ground BC 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 is time-consuming to check θ-subsumption for clauses with many literals. Since a learning algorithm performs numerous coverage tests 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, i.e., naïve, random, and stratified, to generate ground BCs. Given that the BC is built using sampling technique S, we also use S to generate all ground BCs.

Time Complexity. Testing whether a candidate clause (approximately) covers an example using the aforementioned approach is linear in terms of the number of literals in the ground BC of the example [29]. The number of literals in the ground BC is a very small sample of the underlying database. Also, the ground BC are created once for each example at the beginning of learning and are used multiple times for all candidate clauses during generalization. Hence, this approach checks the coverage of of each candidate clause significantly faster than running complex SQL queries with hundreds of joins over the full database.

6 EMPIRICAL STUDY

6.1 Experiment Setup

Data. We run experiments over five datasets. The UW data is explained in Section 1 over which we learn the target relation *advisedBy(stud, prof)*. It 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 *comp* has anti-HIV activity. It has 5 relations, 7.9M tuples, 2K positive and 4K negative examples. IMDb (*imdb.com*) contains information about movies and people who make them. We learn *dramaDirector(dir)*, which shows that *dir* directed a drama movie. It contains 46 relations, 8.4M tuples, 1.8K positive and 3.6K negative examples. The FLT dataset contains information about flights and airports given to us in a funded project. It has three relations, 201K tuples, and 200 positive and 600 negative examples. We were asked to learn the flights with the same source that pass through a given location. SYS contains information about various processes on a server, provided by a private software company. SYS has a single relation of 10.6M tuples with 150 positive and 2000 negative examples. We were asked to learn the patterns of files accesses by malicious processes. SYS has more negative than positive examples due to the rarity of malicious activities. The company selected our relational learning system due to the interpretability of its results.

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 all datasets except for UW and 5-fold 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 built on top of VoltDB, *(voltdb.com)*, a main-memory database system It is shown to be more effective than other available systems [44]. Our implementation is available at *github.com/OSU-IDEA-Lab/AutoBias*.

Methods. We compare AutoBias against several methods. *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. *Aleph* is a popular and public relational learning system, which as opposed to Castor does not use relational database systems. Like 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 [45, 58], which is a popular top-down relational learning algorithm. 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 3. The original databases do not contain INDs. AutoBias calls the IND discovery tools explained in Section 3. The preprocessing step to extract INDs takes 1.2 seconds, 1.4 minutes, 7.8 minutes, 1 minute, and 2.8 minutes over the UW, HIV, IMDb, FLT, and SYS respectively.

Manual Language Bias. The expert wrote 19, 14, 112, 18, and 9 predicate and mode definitions for UW, HIV, IMDb, FLT, and SYS, respectively. The number of predicate and mode definitions for SYS is relatively small due to the information being in a single relation. But, it was still challenging as the expert had to talk to security analysts for a long time to understand the domain and promising patterns and manually inspect thousands of tuples to understand the structure of and the meaning of various constants.

Parameters. We set the constant-threshold hyper-parameter (Section 3.2) to 18% for all datasets. 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 20 tuples per mode for each dataset. In Section 6.3, we evaluate different sampling techniques. We run experiments on a 2.3GHz Intel Xeon E5-2670 processor, running CentOS Linux 7.2 with 500GB memory.

6.2 Approaches to Setting Language Bias

Table 5 illustrates the results of our experiments. 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. Castor is killed by the kernel for other datasets 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 BC contains too many literals, most of which are not useful for learning a definition. *No const.* is the most efficient and obtains competitive F-measure compared to manual tuning and AutoBias over UW. But, it cannot scale over larger datasets and it either takes an extremely long time without producing any results or learns an ineffective definition. The latter is because accurate definitions over many datasets, e.g., IMDb, need constants. Since the top-down learning algorithm used by *Aleph* is generally biased toward learning relatively short clauses, it is faster than other methods over most data. But, Aleph delivers less effective definitions than those by Castor with manual tuning and AutoBias over all datasets.

Over all datasets, except for HIV data, *Manual* results in the most effective definitions. Castor with manual is efficient over all datasets. However, an expert had to spend a significant amount of time tuning the language bias. Further, a non-expert user would not be able to specify this bias.

In general, *AutoBias* is more effective than Castor, No const, Aleph, and almost as effective as manual tuning. Manually written language bias provides a more restricted hypothesis space than the ones generated by AutoBias. Thus, AutoBias has to explore a larger hypothesis space than manual. Nevertheless, the overhead in the running time is none (SYS and UW) or at most about ten minutes (HIV data), which is a reasonable overhead for saving an expert's time and the enterprise's financial resources that pay the machine learning expert. 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.

For the manual approach over IMDb, the expert had to learn the complex schema and relationships between relations. They also had to inspect many tuples from each relation. They progressively developed 112 lines of code to specify predicate and mode definitions. During manual tuning, the expert made syntactic mistakes. We identified these only after running the system and observing the effectiveness of the results. AutoBias did not need most of these steps. AutoBias generated about 30% more predicate and mode definitions, which expands its hypothesis space.

6.3 Sampling Techniques

Table 6 shows the effectiveness and efficiency of learning using different sampling techniques for BC construction and coverage testing. We run random and stratified sampling over each dataset 5 times and computed their average. Generally, random delivers higher efficiency than other methods over IMDb, HIV, and FLT, which confirms that Random selects more promising BCs 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 can explore join paths that lead to all types of elements in a compound.

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. It is also faster than other methods as it does not have their overheads. Naïve is both more efficient and effective than other sampling techniques for SYS. Since SYS is stored in a single large relation, the effective definition does as much of a relational structure compared to the other datasets. It indicates that naïve outperforms other methods over small or single-relation 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 a relational learning setting.

Table 5: Results of different methods of setting language bias (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	>10h	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
FLT	Prec.	-	0	1	0	1
	Recall	-	0	1	0	1
	FM	-	0	1	0	1
	Time	>10h	14m	1m	6s	5.04m
SYS	Prec.	-	-	0.9	0	0.89
	Recall	-	-	0.51	0	0.51
	FM	-	-	0.65	0	0.65
	Time	>10h	>10h	41s	6s	41s

Table 6: Results of 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
FLT	FM	1	1	1
	Time	5.04m	4.96m	4.94m
SYS	FM	0.65	0.39	0.35
	Time	41s	2.19m	6.41m

7 RELATED WORK

Some systems provide users with a graphical representation of the schema so they can specify the mode and predicate definitions easily [23]. Others ask experts to provide examples and advise in the form of logical theories to construct language bias [53]. These systems require heavy experts' intervention. The work in [34] is similar to ours, where the goal is to induce predicate and mode definitions from data. Their algorithm assigns the same type to two attributes if there is an overlap of at least one element. This may deliver a significantly under-restricted search space. Moreover, it does not leverage sampling techniques to improve the efficiency and effectiveness of learning over large databases. An interesting future work is to generate other types of more expressive language biase used in other logical learning settings [32].

Recently, there has been a growing interest in relational learning algorithms that scale to large data in both DB and ML communities [18, 33, 44, 57, 58]. Researchers have used differentiable matrix operations to learn relational models over RDF data [55, 56]. They are limited to datasets with relatively few distinct values, e.g., tens of thousands, and short clauses, e.g., at most 3 literals.

8 CONCLUSION

We have proposed AutoBias, a system that automatically induces the language bias used by relational learning algorithms. Our empirical studies indicate that AutoBias delivers a comparable learning effectiveness to the systems where the language bias is specified by experts.

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] Alexander L. Hayes, Mayukh Das, Phillip Odom, and Sriraam Natarajan. 2017. User Friendly Automatic Construction of Background Knowledge: Mode Construction from ER Diagrams. In *Proceedings of the Knowledge Capture Conference (K-CAP 2017)*. ACM, New York, NY, USA, Article 30, 8 pages. https://doi.org/10.1145/3148011.3148027

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

[30] 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

[31] 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.

[32] Dianhuan Lin. 2013. *Logic programs as declarative and procedural bias in inductive logic programming.* Ph.D. Dissertation. Imperial College London, Department of Computing.

[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] Eric Mccreath and Arun Sharma. 1995. Extraction of Meta-Knowledge to Restrict the Hypothesis Space for ILP Systems. In *Australian Joint Conference on AI*.

[35] Lilyana Mihalkova and Raymond J. Mooney. 2007. Bottom-up learning of Markov logic network structure. In *ICML*.
[36] Tom Mitchell. 1997. *Machine Learning*. McGraw-Hil.
[37] Stephen Muggleton. 1995. Inverse entailment and Progol. *New Generation Computing* 13 (1995), 245–286.
[38] 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.
[39] Stephen Muggleton, Jose Santos, and Alireza Tamaddoni-Nezhad. 2009. ProGolem: A System Based on Relative Minimal Generalisation. In *ILP*.
[40] 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
[41] 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
[42] Frank Olken. 1993. *Random Sampling from Databases*. Ph.D. Dissertation. UC Berkeley.
[43] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & Conquer-based Inclusion Dependency Discovery. *PVLDB* 8 (2015), 774–785.
[44] Jose Picado, Arash Termehchy, and Alan Fern. 2017. Schema Independent Relational Learning. In *SIGMOD Conference*.
[45] J. Ross Quinlan. 1990. Learning Logical Definitions from Relations. *Machine Learning* 5 (1990), 239–266.
[46] Luc De Raedt, David Poole, Kristian Kersting, and Sriraam Natarajan. 2017. Statistical Relational Artificial Intelligence: Logic, Probability and Computation. In *NeurIPS*.
[47] 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
[48] Matthew Richardson and Pedro M. Domingos. 2006. Markov logic networks. *Machine Learning* 62 (2006), 107–136.
[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] Trevor Walker, Ciaran O'Reilly, Gautam Kunapuli, Sriraam Natarajan, Richard Maclin, David Page, and Jude Shavlik. 2011. Automating the Ilp Setup Task: Converting User Advice About Specific Examples into General Background Knowledge. In *Proceedings of the 20th International Conference on Inductive Logic Programming (ILP'10)*. Springer-Verlag, Berlin, Heidelberg, 253–268. http://dl.acm.org/citation.cfm?id=2022735.2022765
[54] 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
[55] 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
[56] 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
[57] Xiaoxin Yin, Jiawei Han, Jiong Yang, and Philip S. Yu. 2004. CrossMine: efficient classification across multiple database relations. *ICDE* (2004), 399–410.
[58] Qiang Zeng, Jignesh M. Patel, and David Page. 2014. QuickFOIL: Scalable Inductive Logic Programming. *PVLDB* 8 (2014), 197–208.
[59] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*.