Learning Over Content-Heterogeneous Relational Databases

Jose Picado
Oregon State University
picadolj@oregonstate.edu

Arash Termehchy
Oregon State University
termehca@oregonstate.edu

ABSTRACT

Given a relational database and training examples for a target relation, relational learning algorithms learn a Datalog program that defines the target relation in terms of the existing relations in the database. The information in a domain is usually spread across several databases, which often represent the same entities under different names. Therefore, learning over multiple databases may result in inaccurate definitions. This type of heterogeneities is also common within a single database due to low quality of the data and inconsistencies in data entry and collection. Currently, users have to spend a great deal of time and effort to resolve these heterogeneities and create a unified and clean database instance to be used for learning. As it is *not* usually clear which values represent the same real-world entity or data item, e.g., whether *J. Smith* represents *John Smith* or *Jeremy Smith*, users may end up with numerous database instances after cleaning and resolving the data values. We propose HDLearn, a relational learning system that performs relational learning directly over heterogeneous databases without any pre-processing step. The user specifies the attributes across relations that contain values that may refer to the same real-world entity or data item through a set of declarative constraints called matching dependencies. HDLearn leverages these dependencies to learn accurate definitions over the heterogeneous data. HDLearn also uses novel sampling techniques that allow it to learn efficiently and output accurate definitions. Our empirical study on real-world datasets indicate that HDLearn learns accurate definitions over large databases efficiently.

PVLDB Reference Format:
Jose Picado and Arash Termehchy. Learning Over Heterogeneous Relational Databases. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Users often use machine learning methods to discover interesting and novel relations over relational databases [12, 21, 42]. For instance, consider the IMDb database (*imdb.com*), which contains information about movies, for which schema fragments are shown in Table 1 (top). Given this database, a user may want to find the definition for the new relation *highGrossing(title)*, which indicates that the movie with a given title is high grossing. Given a relational database and training examples for a new target relation, *relational machine learning* (relational learning) algorithms attempt to learn (approximate) relational definitions of the target relation in terms of existing relations in the database [12, 21]. Definitions are usually restricted to (non-recursive) Datalog programs for efficiency reasons. For instance, the user who wants to learn a definition for the new target relation *highGrossing* using the IMDb database may provide a set of high grossing movies as positive examples and a set of low grossing movies as negative ones to a relational learning algorithm. Given the IMDb database and these examples, the algorithm may learn the following definition:

$$highGrossing(x) \leftarrow movies(y, x, z), mov2genres(y, \text{`}comedy\text{'}),$$
$$mov2releasedate(y, \text{`}June\text{'}, u)$$

which indicates that high grossing movies are often released in June and their genre is *comedy*. Since relational learning algorithms leverage the structure of the database directly to learn new relations and their results are interpretable, they have been widely used in database management, analytics, and machine learning applications, such as query learning and information extraction [12, 21, 27, 2, 29, 42].

Databases, however, often contain heterogeneities in representing data values [14, 20, 8, 17], which may prevent the learning algorithms from finding an accurate definition. In particular, the information in a domain is usually spread across several databases. For example, IMDb does *not* contain the information about the budget or total grossing of the movies. This information is available in another database called Box Office Mojo (BOM) (*boxofficemojo.com*), for which schema fragments are shown in Table 1 (bottom). Thus, to learn an accurate definition for the *highGrossing* relation, the user has to collect (more) data from the BOM

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

Table 1: Schema fragments for the IMDb and Box Office Mojo databases.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

database. For instance, the information about the budget of movies may be helpful in learning an effective definition for the *highGrossing* relation as high grossing movies may have high budgets. Thus, users may integrated these two databases under a unified schema over which to learn the target relation. However, the same entity and/or value may be represented in various forms in the integrated database. For instance, the *titles* of the same movie in IMDb and BOM have different formats., e.g., the title of the movie *Star Wars: Episode IV* is represented in IMDb as *Star Wars: Episode IV- 1977* and in BOM as *Star Wars - IV*. In this case, the learning algorithm *cannot* learn the aforementioned accurate definition for *highGrossing* relation as it will *not* be able to connect the information about the same movie from IMDb and BOM movie relations. Such heterogeneities in representing names of entities and data values appear also in a single data source due to the inconsistencies in data entry and collection among others [8, 17].

Currently, users have to resolve the heterogeneities in representing data values manually and then learn over the clean database. However, it is difficult, time-consuming, and labor-intensive to resolve these heterogeneities over large databases [14, 20]. First, the user has to develop and/or train a matching function that distinguishes and unifies different values that refer to the same entity, apply that function on the database, and materialize the produced instance. Second, a unification may lead to new inconsistencies and opportunities to do more cleaning. For example, after unifying the titles of movies in a database, the user may notice that the names of directors of these movies are different in the database. To keep the database consistent, the user has to unify the names of the directors whose movies have been unified in the recently produced database. This process will be repeated for entities related to directors, e.g., production companies with which a director has worked, and other entities related to the movies until there is no more values to reconcile. Thus, it will take a long time to produce and materialize a clean database instance. Third, the process of unifying values may produce numerous clean database instances as one value may match multiple distinct values [8, 17, 9]. For example, title *Star Wars* may match both titles *Star Wars: Episode IV- 1977* and *Star Wars: Episode III- 2005*. Since we know that the *Star Wars: Episode IV- 1977* and *Star Wars: Episode III- 2005* refer to two different movies, the title *Star Wars* must be unified with only one of them. For each choice, one ends up with a distinct database instance. Since a large database has often many potentially matches, the number of clean database instances may be enormous. It is *not* clear which clean instance one should use to learn an accurate definition for a target relation.

The aforementioned tasks substantially increase the time needed to get insight In fact, most data scientists spend about 80% of their time on data preparation tasks, such as dealing with data heterogeneities [28]. Since users cannot wait for a long time and/or allocate enormous resources to prepare large datasets, researchers have proposed on-demand approaches to data preparation [23, 40, 28, 24]. These methods may reduce the time and effort of data cleaning by preparing only a subset of the data that may be relevant to the task at hand. More importantly, the process data analysis is inherently iterative [24]. Users may start by collecting the data items which they deem relevant to the target relation, clean and prepare them, and learn a definition over the data. If the learned definition is *not* sufficiently accurate, users may look for other data items and repeat this process. An on-demand approach enables users to evaluate the relevance of a dataset to the target relation faster.

In this paper, we build upon the idea of on-demand data analysis and propose a novel learning method that directly learns over the original data without finding and materializing appropriate clean instance(s) of the data. Our approach advances the on-demand approaches as users do *not* have to perform any cleaning and preparation on the underlying data. Instead, users provide only a set of declarative (logical) constraints on the database schema, which determine values of what attribute can meaningfully match and be unified [17, 5, 9, 7, 39, 26, 3, 19, 25]. For instance, the values of *title* attribute in relation *mov2totalGross* in BOM and the ones in the attribute *title* in relation *movies* in IMDb databases may match and be unified. On the other hand, it is *not* meaningful for the values of attributes *gross* in relation *mov2totalGross* and *year* in relation *movies* to be unified as they refer to different types of entities and values. Hence, our method substantially reduces the effort needed to produce clean databases instance(s) for data analysis. Our contributions are as follows:

- Since the representational conflicts in a heterogeneous database are not resolved, the properties of a desirable learned definition over a heterogeneous database are *not* clear. In particular, one heterogeneous database may have several clean instances We introduce and formalize the problem of learning over a relational database with heterogeneities in representing data values (Section 3).
- We propose a novel relational learning algorithm called *HDLearn* that extends current learning methods over databases without heterogeneity to learn over a database with heterogeneous values (Section 4). It leverages the set of declarative constraints on attributes whose values can meaningfully match and unify during learning to learn an effective definition.
- Every learning algorithm chooses the final result based on its coverage of the training data, i.e., roughly speaking, the more (fewer) positive (negative) examples a definition covers the more accurate it is. It has to perform this step for numerous possible definitions. It is challenging to perform these steps effectively over a database without cleaning it as resolving certain entities, e.g., movies, may lead to unifying other entities of the same or other types, e.g., directors. Furthermore, one database may have numerous possible clean versions, We propose an efficient method to compute the coverage of a definition directly over the heterogeneous database without cleaning it (Section 4.2).
- We prove that the definition learned by HDLearn over a heterogeneous database is in fact a compact representation of the definitions learned by HDLearn over its clean version(s) (Section 5). This is a desirable property and indicates that HDLearn leverages all information in the database for learning.
- We use new sampling techniques to scale learning over large databases while keeping it effective (Section **??**).
- We provide an implementation of HDLearn over a main-memory relational database management system and use it to perform an extensive empirical study over real-world datasets and show that HDLearn scales to large databases and learns efficiently over heterogeneous databases.

Missing proofs and additional details about our empirical study are in [].

2. BACKGROUND

2.1 Basic Definitions

We fix two mutually exclusive sets of relation and attribute symbols. A database schema $\mathcal{S}$ is a finite set of relation symbols R_i, $1 \leq i \leq n$. Each relation R_i is associated with a set of attribute symbols denoted as $R_i(A_1, \ldots, A_m)$. We denote the domain of values for attribute A as $dom(A)$. Each database instance I of schema $\mathcal{S}$ maps a finite set of tuples to every relation R_i in $\mathcal{S}$. Each tuple t is a function that maps each attribute symbol in R_i to a value from its domain. We denote the value of the attribute X of tuple t in the database I by $t^I[X]$. An *atom* is a formula in the form of $R(u_1, \ldots, u_n)$, where R is a relation symbol and $u_1, \ldots, u_n$ are *terms*. Each term is either a variable or a constant, i.e., value. A *ground atom* is an atom that only contains constants. A *literal* is an atom, or the negation of an atom. A *Horn clause* (clause for short) is a finite set of literals that contains exactly one positive literal. A *ground clause* is a clause that only contains ground atoms. Horn clauses are also called Datalog rules (without negation) or conjunctive queries. A *Horn definition* is a set of Horn clauses with the same positive literal, i.e., non-recursive Datalog program or union of conjunctive queries. A literal L_i in a clause $H \leftarrow L_1, \ldots, L_n$ is *head-connected* if and only if at least one variable in L_i appears either in H or in a body literal L_j, where $1 \leq j < i$.

2.2 Relational Learning

In this section, we present the basic concepts of relational learning over databases without any heterogeneity and matching dependencies [12, 21]. Relational learning algorithms learn first-order logic definitions from an input relational database and training examples. Training examples E are usually tuples of a single *target relation*, and express positive (E_+) or negative (E_-) examples. The input relational database is also called *background knowledge*. The *hypothesis space* is the set of all possible first-order logic definitions that the algorithm can explore. It is usually restricted to Horn definitions to keep learning efficient. Each member of the hypothesis space is a *hypothesis*. Clause C *covers* an example e if $I \wedge C \models e$, where $\models$ is the entailment operator, i.e., if I and C are true, then e is true. Definition H covers an example e if any of the clauses in H covers e. The goal of a learning algorithm is to find the definition, i.e., hypothesis, in the hypothesis space that covers all positive and the fewest negative examples as possible.

Most relational learning algorithms follow a covering approach, depicted in Algorithm 1 [34, 32, 30, 42, 33]. They construct one clause at a time using the *LearnClause* function. If the clause satisfies a criterion, e.g., covers at least certain fraction of the positive examples and does *not* cover certain fraction of negative ones, they add the clause to the learned definition and discards the positive examples covered by the clause. They stop when all positive examples are covered by the learned definition.

Algorithm 1: Covering approach algorithm.

Input : Database instance I, examples E
Output: 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 ∪ C
        U = U − {e ∈ U | H ∧ I ⊨ e}
return H
```

2.3 Matching & Cleaning

2.3.1 Matching Dependencies

Learning over databases with heterogeneity in representing values may deliver inaccurate answers as same entities and values may be represented under different names. Thus, one must resolve these representational differences to produce a high-quality database to learn an effective definition. If one wants to match and resolve values in a couple of attributes, one may use a supervised or unsupervised matching function to identify and unify their potential matches according to the domain of those attributes [14, 20]. For example, given that the domain of attributes is a set of strings, one may use string similarity functions, such as edit distance to find potential matches. Users develop and use matching and resolution rules and functions based on their domain knowledge. For example, consider relation *Employee(id, name, home-phone, address)*. According to her domain knowledge, the user know that if phone numbers of two tuples are sufficiently similar, e.g., *001-333-1020* and *333-1020*, their addresses must be equal. Knowing this rule, the user may manipulate the database to ensure that all tuples with a similar phone number have equal addresses.

There may be a large number of matching rules over several sets of attributes in a large relational database [17, 5, 9, 7, 39, 26, 3, 19, 25]. Furthermore, these rules may interact with each other. For example, consider again the relation *Employee(id, name, home-phone, address)*. Assume that the user also knows if two tuples have sufficiently similar addresses, e.g., *1 Main St., NY* and *1 Main Street, New York*, and names, they must have the same values for attribute *id*. Now, consider two tuples whose names and home phone numbers are sufficiently similar but their addresses are *not*. According to this rule, one *cannot* unify the ids of these tuples. But, after applying the rule mentioned in the preceding paragraph on the phone number and address, their addresses become sufficiently similar. Therefore, one can apply the second rule to unify the values of *id* in these tuples. the abundance of rules to match values and their interactions and dependencies call for a more formal approach to data cleaning where one can guarantee that all matching and resolution rules have been applied to the original data.

The database community has proposed declarative matching and resolution rules to express the domain knowledge about matching and resolution [17, 5, 9, 7, 39, 26, 3, 19, 25]. *Matching dependencies (MD)* is a popular type of such declarative rules, which provides a simple yet powerful method of expressing domain knowledge on matching values [17, 8, 6]. Let $\mathcal{S}$ be the schema of the original database and R_1 and R_2 two relations in $\mathcal{S}$. Attributes A_1 and A_2 from relations R_1 and R_2, respectively, are comparable if $dom(A_1) = dom(A_2)$. Given two pairs of pairwise comparable attributes

A_1, A_2 and B_1, B_2 from relations R_1 and R_2, respectively, an MD ϕ in $\mathcal{S}$ is a sentence of the form $R_1[A_1] \approx R_2[A_2] \rightarrow R_1[B_1] \rightleftharpoons R_2[B_2]$, where $\approx$ is a similarity operator and $R_1[B_1] \rightleftharpoons R_2[B_2]$ indicates that the values of $R_1[B_1]$ and $R_2[B_2]$ refer to the same value, i.e., are interchangeable. Intuitively, the aforementioned MD says that given the values of $R_1[A_1]$ and $R_2[A_2]$ are sufficiently similar, the values of $R_1[B_1]$ and $R_2[B_2]$ are essentially different representations of the same value. For example, consider again the database that contains relations from *IMDb* and *BOM* whose schema fragments are shown in Tables **??** and **??**, respectively. According to our discussion in Section 1, one can define the following MD: *movies[title]* $\approx$ *highBudgetMovies[title]* $\rightarrow$ *movies[title]* $\rightleftharpoons$ *highBudgetMovies[title]*.

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

2.3.2 Stable Instances

Given an input database with MDs on its relations, one must enforce the MDs to generate a high-quality database. Let t_1 and t_2 belong to R_1 and R_2 in database I of schema $\mathcal{S}$, respectively, such that $t_1^I[A_1] \approx t_2^I[A_2]$. To enforce the MD $\phi : R_1[A_1] \approx R_2[A_2] \rightarrow R_1[B_1] \rightleftharpoons R_2[B_2]$ on I, one makes the values of $t_1^I[B_1]$ and $t_2^I[B_2]$ identical as they actually refer to the same value [17, 8]. For example, if attributes B_1 and B_2 contain titles of movies, one unifies both values *Star Wars - 1977* and *Star Wars* to *Star Wars. Episode IV - 1977* as it deems this value as the one to which $t_1^I[B_1]$ and $t_2^I[B_2]$ refer. The following definition formalizes the concept of applying an MD to the tuples t_1 and t_2 on I.

DEFINITION 2.1. *Database I' of schema $\mathcal{S}$ is the immediate result of enforcing MD ϕ on t_1 and t_2 in I, denoted by $(I, I')_{[t_1,t_2]} \models \phi$ if*

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

One may define a matching function over some domains to map the values that refer to the same value/ entity to the correct value/ entity in the stable instance. It may, however, be difficult to define such a function in many cases due to the lack of information. For example, let B_1 and B_2 in Definition 2.1 contain information about names of people and $t_1^I[B_1]$ and $t_2^I[B_2]$ have values *J. Bryan Smith* and *John B. Smith*, respectively, which according to an MD refer to the same actual name. It is *not* clear how to compute this name using the values of $t_1^I[B_1]$ and $t_2^I[B_2]$. Thus, we know that after enforcing ϕ, the values of $t_1^{I'}[B_1]$ and $t_2^{I'}[B_2]$ will be identical, but we do *not* know their exact values. Because we aim at developing learning algorithms that are efficient and effective over databases from various domains, we do *not* fix any matching method in this paper. We assume that matching every pair of values a and b in the database creates a fresh and new value denoted as $< a, b >$.

Given the database I of schema $\mathcal{S}$ with set of MDs Σ, I is *stable* iff $(I, I)_{[t_1,t_2]} \models \phi$ for all $\phi \in \Sigma$ and all tuples $t_1, t_2 \in I$. In a stable database instance, all values that represent the same data item according to the database MDs are assigning equal values. Thus, it does *not* have the heterogeneities that may reduce the effectiveness of learning and analyzing databases. Given a database I with set of MDs Σ, one can produce a stable instance for I by starting from I and iteratively applying each MD in Σ according to Definition 2.1 to the output of the previous application finitely many times. In other words, let $I, I_1, \ldots, I_k$ denote the sequence of databases produced by applying MDs according to Definition 2.1 starting from I such that I_k is stable. We say that (I, I_k) satisfy Σ and denote it as $(I, I_k) \models \Sigma$. Each database may have (finitely) many stable instances depending on the order of applications of MDs [17, 8].

EXAMPLE 2.2. *Let (10,'Star Wars: Episode IV- 1977', 1977) and (40,'Star Wars: Episode III- 2005', 2005) be tuples in relation movies and ('Star Wars') be a tuple in relation highBudgetMovies whose schemas are shown in Tables* **??** *and* **??**, *respectively. Consider MD movies[title]* $\approx$ *highBudgetMovies[title]* $\rightarrow$ *movies[title]* $\rightleftharpoons$ *highBudgetMovies[title]. Let 'Star Wars: Episode IV- 1977'* $\approx$ *'Star Wars' and 'Star Wars: Episode III- 2005'* $\approx$ *'Star Wars' be true. Since the movies with titles 'Star Wars: Episode IV- 1977' and 'Star Wars: Episode III- 2005' are different movies with distinct titles, one can unify the title in the tuple ('Star Wars') in highBudgetMovies with only one of them in each stable instance. Each unification alternative leads to a distinct stable instance.*

Number of stable instances of a database generally grows exponentially with the number of matches and unification applications in the database. For example, in Example 2.2, if relation *highBudgetMovies* has a hundred titles that each matches and unified with at least two titles in relation *movies*, the database will have around 2^100 stable instances. Since there are often many matches and unification applications in a large database, one may have to generate numerous stable and clean databases. To produce only one stable instance, one may assume that all distinct unified values for a given value in the database represent the same value. For example, in Example 2.2, if one assumes that all movies with prefix *Black Sheep* represent the same movie, one can unify the titles of all movies to the same value. Nevertheless, this approach often violate the semantics of the underlying database and adds misleading information to it. For example, in Example 2.2, this approach leads to assuming that movies with titles 'Star Wars: Episode IV- 1977' and 'Star Wars: Episode III- 2005' have the same titles, which is obviously incorrect and misleading. This approach may also violate the integrity constraints in the database. For example, if matches and unifications are between key attributes, this approach will violate the key or functional dependency constraint by adding duplicate keys to a relation and makes the database inconsistent.

3. LEARNING A DEFINITION OVER HETEROGENEOUS DATA

3.1 Approaches to Learning Over Heterogeneous Data

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

However, this method is *not* desirable and/or feasible for learning over large databases. As a large database may have numerous stable instances, it takes a great deal of time and storage to compute and materialize all of them. Moreover, running the learning algorithm we have to run the learning algorithm once for each stable instance, which may takes an extremely long time. More importantly, as the learning has been done separately over each stable instance, it is *not* clear whether the final definition is sufficiently effective considering the information of all stable instances. For example, let database I have two stable instances I_1^s and I_2^s over which the aforementioned approach learns definitions H_1 and H_2, respectively. H_1 and H_2 must cover a relatively small number of negative examples over I_1^s and I_2^s, respectively. However, H_1 and H_2 may cover a lot of negative examples over I_2^s and I_1^s, respectively. Thus, the disjunction of H_1 and H_2 will *not* be effective considering the information in I_1^s and I_2^s. Hence, it is *not* clear whether the disjunction of H_1 and H_2 is the definition that covers the most positive and the least negative examples over I_2^s and I_1^s. Finally, it is *not* clear how to encode and represent usably the final result as we may end up with numerous different definitions each of which is accurate over one stable instance. One may end up with hundreds or thousands of learned definitions. Thus, learning over each stable instance may take a very long time, *not* return an effective definition that considers information of all stable instances, and/or will be hard to use.

Another approach is to consider only the information shared among all stable instances for learning. The resulting definition will cover the most positive and the least negative examples considering the information common among all clean instances. This idea has been used in the context of query answering over inconsistent data [4, 8]. However, this approach may lead to ignoring many positive and negative examples as their connections to other relations in the database may *not* be present in all stable instances. For example, consider the tuples in relations *movies* and *highBudgetMovies* in Example 2.2. The training example *('Star Wars')* has different values in different stable instances of the database, therefore, it will be ignored. It will also be connected to two distinct movies with vastly different properties in each instance. Nevertheless, the training examples are very important in delivering an effective result and are usually costly to attain. In a sufficiently heterogeneous database, most or all positive and negative examples will *not* have the same information among all stable instances. The learning algorithm may simply learn an empty definition in these cases.

Thus, we hit a middle-ground. We follow the approach of learning directly over the original database. But, we also give the language of definitions and semantic of learning enough flexibility to take advantage of as much as (training) information as possible. If one increases the expressivity of the language, learning and checking coverage for each clause may become inefficient, e.g., disjunctive Datalog [15]. We ensure that the added capability to the language of definitions is minimal so learning remains efficient. In this section, we present our modifications to the hypothesis space and semantic of learning.

3.2 Representing Heterogeneity in Definitions

We represent the heterogeneity of the underlying data in the learned definition to increase its coverage and avoid the problem of ignoring many training examples. To achieve this, we increase the hypothesis space of relational learning algorithms over schema $\mathcal{S}$ by introducing a fresh and unique (built-in) relation symbol per each MD in $\mathcal{S}$. These symbols do *not* belong to $\mathcal{S}$, and are called *matching relations*. More precisely, given schema $\mathcal{S}$ and MD ϕ in $\mathcal{S}$, we add relation symbol m_ϕ with arity two to the set of relation symbols used by the Datalog definitions in the hypothesis space. A literal with a matching relation symbol is a *matching literal*. Consider again the database created by integrating IMDb and BOM datasets whose schema fragments are in Tables **??** and **??** with MD ϕ : *movies[title]* $\approx$ *highBudgetMovies[title]* $\rightarrow$ *movies[title]* $\rightleftharpoons$ *highBudgetMovies[title]*. We may have the following Datalog definition for the target relation *highGrossing*.

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

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

We call a clause (definition) *stable* if it does *not* have any matching literal. Each clause (definition) with matching literals represents a set of stable clauses (definitions). We convert a clause with matching literals to a set of stable clauses by iteratively applying the MDs that correspond to each matching literal and eliminating the matching literal from the clause, similar to the process of applying MDs to a database instance described in Section 2.3. Given a clause C and an MD ϕ, to apply ϕ to C, we do the following. For each matching literal $m_\phi(x,y)$ in C, we create a new variable $< x, y >$. We then replace every occurrence of x and y in C with $< x, y >$. Finally, we remove the original matching literal $m_\phi(x,y)$ from C. The algorithm progressively applies MDs to the created clause until no matching literal is left in the clause. Similar to the one explained in Section 2.3, this algorithm is guaranteed to terminate. The resulting set is called the *stable clauses* of the input clause.

EXAMPLE 3.1. *Consider the following clause over the movie database of IMDb and BOM.*

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

The application of MD $\phi : highGrossing[title] \approx movies[title] \rightarrow highGrossing[title] \rightleftharpoons movies[title]$*, unifies variables* t *and* x *to variable* $< x, t >$ *and generates the following clause:*

$$hihighGrossing(< x, t >) \leftarrow movies(y, < x, t >, z), \\ mov2genres(y, \text{`comedy'}), highBudgetMovies(< x, t >).$$

After each application of an MD to a clause, the modified variables will *not* be similar to any term in the clause, which is equivalent to the same way we treat the result of application of MDs on databases in Section 2.3. This is because we do *not* have enough information to declare the variable/constant created after matching equal to any other variable/constant. Similar to the application of MDs to a database instance, the application of MDs to a clause may generate multiple stable clauses.

EXAMPLE 3.2. *Consider a target relation* $T(A)$*, an input database with schema* $\{R(B), S(C)\}$*, and MDs* $\phi_1 : T[A] \approx R[B] \rightarrow T[A] \rightleftharpoons R[B]$ *and* $\phi_2 : T[A] \approx S[C] \rightarrow T[A] \rightleftharpoons S[C]$*. The definition* $H : T(x) \leftarrow R(y), m_{\phi_1}(x, y), S(z), m_{\phi_2}(x, z)$ *over this schema has two stable definitions:* $H'_1 : T(< x, y >) \leftarrow R(< x, y >), S(z)$ *and* $H'_2 : T(< x, z >) \leftarrow R(y), S(< x, z >)$*.*

As Example 3.2 illustrates using matching literals enable us to provide a compact representation of multiple learned clauses/definitions where each may explain the patterns in the training data in some stable instances of the input database. Given an input definition H, the *stable definitions* of H are a set of definitions where each definition contains exactly one stable clause per each clause in H.

3.3 Coverage over Heterogeneous Data

Next, we define the coverage of positive examples for definitions that may contain matching literals.

DEFINITION 3.3. *A definition* H *covers a positive example* e *with the regard to database* I *iff every stable definition of* H *covers* e *in some stable instance of* I*.*

EXAMPLE 3.4. *As an example consider again the schema, MDs, and definition* H *in Examples 3.2 and the database of this schema with training example* $T(a)$ *and tuples* $\{R(b), S(c)\}$*. Assume that* $a \approx b$ *and* $a \approx c$ *are true. The database has two stable instances* $I'_1 : \{T(< a, b >), R(< a, b >), S(c)\}$ *and* $I'_2 : \{T(< a, c >), R(b), S(< a, c >)\}$*. Definition* H *covers the single training example in the original database according to Definition 3.3 as its stable definitions* H'_1 *and* H'_2 *cover the training example in stable instances of* I'_1 *and* I'_2*, respectively.*

Definition 3.3 provides a more flexible semantic than considering only the common information between all stable instances as described in Section 3.1. The latter semantic considers that the definition H covers a positive example if it covers the example in all stable instances of a database. As explained in Section 3.1, this approach may lead to ignoring many if not all examples for learning. The semantic of coverage in Definition 3.3 is similar to the one of learning each definition separately over each stable instance and then delivering their union, i.e., disjunction. In this case, each definition or clause presented in the final one covers some positive examples in some stable instances of the database.

To compute the coverage of a clause C for a set of positive examples, learning algorithms simply check the coverage of the clause for each example in the set, union the covered examples, and report their sum. This approach can be safely applied for the case where C is stable. However, if C is *not* stable, this method deliver a clause that does *not* cover sufficiently many positive examples in any stable instance. For example, consider a database with one hundred stable instance and a set of 100 positive examples. Let a each stable clause of a clause C cover only one example over each stable instance of the database. Obviously, C is *not* effective on any of these instances as its stable clauses cover an extremely small number of positive examples over each stable instance. This method, however, declares that C covers all the positive examples and is effective. Thus, a more reasonable approach is to ensure that each stable clause of C also cover sufficiently many positive examples over their corresponding stable instance(s). Given database I, stable clause C, and example e, let $\mathcal{I}_{C,e}$ be the set of stable instances of I on which C covers e.

DEFINITION 3.5. *Given a database* I*, clause* C*, and set of positive examples* E*,* C *covers* E *iff* C *covers each* $e \in E$ *and for every stable clause* C_s *of* C *and every pair of examples* $e, g \in E$*, we have* $\mathcal{I}_{C_s,e} = \mathcal{I}_{C_s,g}$*.*

Definition 3.5 guarantees that the number of examples covered by a clause is correctly computed in each stable instance of the input database. The definition naturally extends to Datalog definitions.

One may also use Definition 3.3 to determine the coverage of a negative example. That is, one may consider a definition H *not* to cover a negative example w.r.t. I if at least one of its stable definitions does *not* cover the example in all stable instances of I. However, in this case, other stable definitions of H may cover the negative examples in all or some stable instances of I. Thus, it will *not* be clear whether H is effective considering the information of all stable instances as explained in Section 3.1. Thus, we follow a stronger semantic for negative coverage as follows.

DEFINITION 3.6. *A definition* H *covers a negative example* e *with the regard to database* I *iff none of the stable definitions of* H *cover* e *in any stable instance of* I*.*

As explained in Section 3.1, our semantic of coverage based on Definitions 3.3 and 3.6 provides a middle-ground between the approaches of using only the information common between all stable instances and learning over each stable instance separately. It avoids ignoring too many examples and also ensures that the learned definition does *not* cover too many negative example overall.

4. HDLEARN: A LEARNING ALGORITHM FOR HETEROGENEOUS DATA

There are, generally speaking, two types of relational learning algorithms. In the top-down approach [34, 36, 42], the *LearnClause* function in Algorithm 1, searches the hypothesis space from general to specific, by using a refinement operator that is generally adding a new literal to the current clause. It starts with an empty clause, iteratively adds some literals to it, evaluates the resulting clause(s) over the training examples and database, and stops as soon as the clause satisfies certain criteria, e.g., covers the most positive

and least negative examples. This method is challenging to use for learning over heterogeneous data, as it has to evaluate numerous clauses, i.e., queries, over the original (dirty) dataset. Since the dataset may have many stable solutions, query answering is shown to be generally difficult and inefficient over a relatively large and heterogeneous dataset [8].

The second approach for relational learning algorithms is bottom-up, in which the algorithm starts by exploring the patterns and clauses available in the data and then generalizes them to cover the training examples [30, 32]. More specifically, in these algorithms, the *LearnClause* function has two steps. It first builds the most specific clause in the hypothesis space that covers a given positive example, called a *bottom-clause*. Then, it generalizes the bottom-clause to cover as most positive and as fewest negative examples as possible. In this section, we propose a bottom-up algorithm called HDLearn for learning over heterogeneous data efficiently. To do so, HDLearn integrates the input MDs into the learning process. It also uses novel techniques to pick the most accurate clauses (and definitions) without computing the stable instances of the data. Because it is guided by the training data, HDLearn automatically determines whether and which databases one has to integrate to learn an accurate definition for the target concept. Next, we explain the two main steps of the algorithm in detail.

4.1 Bottom-clause Construction

A *bottom-clause* C_e associated with an example e is the most specific clause in the hypothesis space that covers e. Let I be the input database of schema $\mathcal{S}$, and Σ be the input matching dependencies. The bottom-clause construction algorithm consists of two phases. First, HDLearn finds all the information in I relevant to e. The information relevant to example e is the set of tuples $I_e \subseteq I$ that are connected to e. A tuple t is connected to e if we can reach t using a sequence of (similarity) search operations, starting from e. Given the information relevant to e, HDLearn creates the bottom-clause C_e.

EXAMPLE 4.1. *Given example highGrossing(Superbad), the database in Table 2, and MD*

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

HDLearn finds the relevant tuples movies(m1, Superbad (2007), 2007), mov2genres(m1, comedy), mov2countries(m1, c1), englishMovies(m1), mov2releasedate(m1, August, 2007), and countries(c1, USA). As the movie title in the training example, e.g., Superbad, does not match with the movie title in the movies relation, e.g., Superbad (2007), the tuple movies(m1, Superbad (2007), 2007) is obtained through a similarity search according to MD ϕ. The other tuples are obtained through exact search.

To find the information relevant to e, HDLearn uses Algorithm 2. HDLearn maintains a set M that contains all seen constants. Let $e = T(a_1, \ldots, a_n)$ be a training example. First, HDLearn adds $a_1, \ldots, a_n$ to M (line 3). These constants are values that appear in tuples in I. Then, HDLearn searches all tuples in I that contain at least one constant in M and adds them to I_e (lines 5-11). For exact search, HDLearn uses simple *select* operations (line 7). For similarity search, HDLearn uses MDs in Σ. If M contains

movies(m1,Superbad (2007),2007)	mov2genres(m1,comedy)
movies(m2,Zoolander (2001),2001)	mov2genres(m2,comedy)
movies(m3,Orphanage (2007),2007)	mov2genres(m3,drama)
mov2countries(m1,c1)	countries(c1,USA)
mov2countries(m2,c1)	countries(c2,Spain)
mov2countries(m3,c2)	englishMovies(m1)
mov2releasedate(m1,August,2007)	englishMovies(m2)
mov2releasedate(m2,September,2001)	spanishMovies(m3)

Table 2: Example database.

constants in some relation R_i and given an MD $\phi \in \Sigma$, $\phi : R_i[A] \approx R_j[B] \rightarrow R_i[A] \rightleftharpoons R_j[B]$, HDLearn performs a similarity search over $R_j[B]$ to find relevant tuples in R_j, denoted by $\psi_{B \approx M}(R_j)$ (lines 8-9). For each new tuple in I_e, the algorithm extracts new constants and adds them to M (lines 10-11). The algorithm repeats this process for a fixed number of iterations d (line 4).

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 ∈ I do
6  |  |  foreach attribute A in R do
7  |  |  |  I_R = σ_{A∈M}(R)
8  |  |  |  if ∃ MD φ ∈ Σ,
   |  |  |  |  φ : S[B] ≈ R[A] → S[B] ⇌ R[A] then
9  |  |  |  |  I_R = I_R ∪ ψ_{A≈M}(R)
10 |  |  |  foreach tuple t ∈ I_R do
11 |  |  |  |  add t to I_e and constants in t to M
12 C_e = create clause from e and I_e
13 return C_e
```

To create the bottom-clause C_e from I_e, HDLearn first maps each constant in M to a new variable. It creates the head of the clause by creating a literal for e and replacing the constants in e with their assigned variables. Then, for each tuple $t \in I_e$, HDLearn creates a literal and adds it to the body of the clause, replacing each constant in t with its assigned variable. If there is an input MD $\phi : R_i[A] \approx R_j[B] \rightarrow R_i[A] \rightleftharpoons R_j[B]$, and there is a tuple $t' \in R_i$, then I_e may contain a tuple t obtained through similarity search. In this case, we add a *matching literal* $m_\phi(v_1, v_2)$, where v_1 and v_2 are the variables assigned to $t'[A]$ and $t[B]$, respectively. A matching literal indicates that the two values represented by the variables in the literal are interchangeable, according to the MD.

EXAMPLE 4.2. *Given the relevant tuples found in Example 4.1, HDLearn creates the following bottom-clause:*

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

4.2 Generalization

After creating the bottom-clause C_e for example e, HDLearn generalizes C_e iteratively. To generalize C_e, HDLearn randomly picks a subset $E^s_+ \subseteq E_+$ of positive examples. For each example e' in E^s_+, HDLearn generalizes C_e to generate a candidate clause C', which is more general than C_e and covers e'. It does so by removing the *blocking literals*. Let $C_e = T \leftarrow L_1, \cdots, L_n$ be the bottom-clause. L_i is a *blocking literal* iff i is the least value such that for all substitutions θ where $e' = T\theta$, the clause $(T \leftarrow L_1, \cdots, L_i)\theta$ does not cover e' [32]. From the set of generalized clauses, HDLearn selects the highest scoring candidate clause. The score of a clause is the number of positive examples covered by the clause minus the number of negative examples covered by the clause. It then repeats this process with the selected clause until the clause cannot be improved.

EXAMPLE 4.3. *Consider the bottom-clause C_e in Example 4.2 and positive example $e' = highGrossing(Zoolander)$. To generalize C_e to cover e', HDLearn drops the literal $mov2releasedates(y, \text{'}August\text{'}, u)$ because the movie Zoolander was not released in August.*

A matching literal can also be a blocking literal.

EXAMPLE 4.4. *Consider again the bottom-clause C_e in Example 4.2 and positive example $e'' = highGrossing(Inception)$. The movie with title 'Inception' does not appear in the movies relation. Therefore, the first blocking literal in C_e is $m_\phi(x, t)$, which is in the join path between $highGrossing(x)$ and $movies(y, t, z)$.*

HDLearn checks whether a candidate clause covers training examples in order to find blocking literals in a clause. It also computes the score of a clause by computing the number of training examples covered by the clause. Coverage tests dominate the time for learning [12]. One approach to perform a coverage test is to transform the clause into a SQL query and evaluate it over the input database to determine the training examples covered by the clause. However, since bottom-clauses over large databases normally have many literals, e.g., hundreds, the SQL query will involve long joins, making the evaluation extremely slow. Furthermore, it is challenging to evaluate clauses using this approach over heterogeneous data [8]. Moreover, it is *not* clear how to evaluate clauses that contain matching literals.

Therefore, HDLearn uses an approach called θ-subsumption. Assume that clause C does not contain matching literals, i.e., C is a stable clause. Clause C *θ-subsumes* clause G, denoted by $C \subseteq_\theta G$, iff there is some substitution θ such that $C\theta \subseteq G$ [12, 1]. $C\theta \subseteq G$ means that the result of applying substitution θ to clause C is a subset of clause G. The relation of θ-subsumption is both sound and complete for Horn definitions without functions, which are the focus of this paper, i.e., $C\theta \subseteq G$ iff $C \models G$ [1]. To evaluate whether a clause C covers an example e over database I, we first build a ground bottom-clause G_e for e in I, i.e., a bottom-clause for e without replacing constants with variables. Then, we check whether $C \subseteq_\theta G_e$, which in turn indicates whether $C \wedge I \models e$. This approach is used by other learning algorithms [32, 33]. Ideally, G_e must have one literal per each tuple in the database that is connected to e through some joins, otherwise, the θ-subsumption test may declare that C does *not* cover e when C actually covers e. Ground bottom-clauses may be very large over large databases. The θ-subsumption is expensive over large bottom-clauses. We will discuss how to handle these cases in Section 6.1.1.

Assume that clause C is stable, but ground bottom-clause G_e of e is not, i.e., it contains matching literals. If clause C covers G_e, then C covers e in all the stable instances of the input database. Now, assume that both clause C and ground bottom-clause G_e are not stable. This means that there are possibly multiple stable clauses that can be derived from C and G_e. According to Definition 3.3, clause C covers example e iff every stable clause derived from G_e is covered by at least one stable clause derived from C. Following this idea, we define θ-subsumption between potentially non-stable clauses.

DEFINITION 4.5. *Given clauses C and G over schema $\mathcal{S}$, clause C θ-subsumes G if and only if each stable clause of C θ-subsumes at least one stable clause of G and every stable clause of G is θ-subsumed by a stable clause of C.*

According to Definition 4.5, given training example e, to check whether C θ-subsumes G_e, one has to enumerate and check θ-subsumption of almost every pair of stable clauses of C and G_e in the worst case. Since both (ground) bottom-clauses normally contain many literals and θ-subsumption is NP-hard [1], this method is time-consuming. More importantly, because the learning algorithm performs numerous coverage tests, learning a definition may be extremely inefficient. Actually, we can check whether C θ-subsumes G_e in a much more efficient way and without enumerating all stable clauses of C and G_e. We consider matching literals in C and G_e that correspond to the same MD as the same relation symbol. To find a substitution θ for C such that $C\theta \subseteq G_e$, we treat matching literals in C and G_e as normal literals. If such substitution exists, then C θ-subsumes G_e, hence C covers e.

EXAMPLE 4.6. *Consider clause C*

$$highGrossing(x) \leftarrow movies(y, t, z), m_\phi(x, t), mov2genres(y, comedy).$$

Consider positive example $e' = highGrossing(Zoolander)$ and ground bottom-clause $G_{e'}$

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

C θ-subsumes $G_{e'}$ using substitution $\theta = \{x = Zoolander, t = Zoolander(2001), y = m2, z = 2001\}$. Therefore, C covers e'.

The following theorem shows that the aforementioned algorithm correctly detects θ-subsumption between clauses.

THEOREM 4.7. *Given a clauses C and G over schema $\mathcal{S}$, clause C θ-subsumes G if and only if the aforementioned algorithm finds a substitution θ such that $C\theta \subseteq G$.*

PROOF. Assume that there is a substitution θ such that $C\theta \subseteq C'$. Thus, each matching literal of C, such as $m_\phi(x, y)$ is mapped and unified with a matching literal $m_\phi(x', y')$ in C'. Let C_ϕ and C'_ϕ be the results of applying $m_\phi(x, y)$ and

$m_\phi(x', y')$ to C' and C', respectively. We show that the applications of mapped matching literals in these clauses preserve the substitution, i.e., $C_\phi\theta \subseteq C'_\phi$. Applying a matching literal does *not* eliminate any non-matching literal from a clause and only replaces some of their variables with new variables. Let C_ϕ and C'_ϕ be created by applying $m_\phi(x, y)$ and $m_\phi(x', y')$ to $R(\bar{x}), S(\bar{y})$ and $R(\bar{x'}), S(\bar{y'})$, respectively. The application of $m_\phi(x, y)$ replaces x and y in $R(\bar{x})$ and $S(\bar{y})$ with a new variable $< x, y >$. Similarly, by applying $m_\phi(x', y')$, we replace x' and y' in $R(\bar{x'})$ and $S(\bar{y'})$ with $< x', y' >$. The applications of $m_\phi(x, y)$ and $m_\phi(x', y')$ do *not* modify the literals that do *not* share any variable with them. Thus, we have $C_\phi\theta \subseteq C'_\phi$. Furthermore, as we assume that matching functions are *not* similarity preserving, subsequent applications of the matching literals other than $m_\phi(x, y)$ and $m_\phi(x', y')$ do *not* modify the variables introduced by applying these matching literals. Thus, the substitution θ is preserved over the application of each matching literal in C and its corresponding one in C'. After exhausting the application of all matching literals in C and their corresponding ones in C', C' may still contain matching literals. However, applying these literals to C' retains the substitution between the stable clause of C and subsequent results of application of produced stable clauses of C'. The necessity of the proposition is proved by reversing the aforementioned steps. □

Given bottom-clause C and a set of examples E, according to Definition 3.5, it seems that one *cannot* use the simple method of computing the coverage of C for each example in E and summing up the results to calculate the coverage of C over E. However, the following proposition shows that if one uses the aforementioned subsumption method to check the coverage of C for each example, this simple method returns the correct coverage value for C. Therefore, our algorithm is able to compute the coverage of a non-stable clause over a set of examples without any extra overhead.

THEOREM 4.8. *Given a database I, set of examples E, and clause C, let C θ-subsume G_{e_1} and G_{e_2}, which are ground bottom-clauses for $e_1, e_2 \in E$, respectively. Then, for each stable clause C_s of C, we have $\mathcal{I}_{C_s,e_1} = \mathcal{I}_{C_s,e_2}$*

PROOF. The proposition holds trivially if G_{e_1} and G_{e_2} do *not* share any normal literal. Now, assume that G_{e_1} and G_{e_2} share literal $R(a, b)$. For the sake of simplicity, let $R(a, b)$ be the only literal shared between the bodies of G_{e_1} and G_{e_2}. Our proof extends to other cases. According to the algorithm of creating ground bottom-clauses, all literals in G_{e_1} that have a constant in common with $R(a, b)$ or their corresponding tuples in the database are connected to the tuple of $R(a, b)$ in the data through some join paths will appear also in G_{e_2}. Let us call this set of literals L. Furthermore, according to the algorithm of creating ground bottom-clauses, all matching literals that have some constants in common with a literal in L appear in both G_{e_1} and G_{e_2}. L does *not* share any constant with other literals that do *not* belong to L neither in G_{e_1} nor G_{e_2}. Similarly, the matching literals that share some constants with some literal in L do *not* share any constant with the ones that do *not* belong to L in G_{e_1} nor G_{e_2}. Thus, the subset of the substitution θ from C to L and its related matching literals in G_{e_1} and G_{e_2} does not share any variable/ constant in the domain, i.e., variables/constants in C, and range, i.e., constants in L and its matching literals, with the rest of θ. We call this subset θ_L, which is used for both G_{e_1} and G_{e_2}. Let G_{e_1} have a stable clause $G^s_{e_1}$ that is subsumed by a stable clause of C called C^s. Let the literals of L in $G^s_{e_1}$ be L^s. Since C^s subsumes $G^s_{e_1}$, a subset of literals in C^s, called C^s_L subsumes L^s. L^s must also appear in some stable clause of G_{e_2} called $G^s_{e_2}$. L^s is created from L in both G^s_1 and $G^s_{e_2}$ by applying the same set of matching literals in the same order. Since $G^s_{e_1}$ and $G^s_{e_2}$ do *not* share any literal in addition to the ones in L^s, if C^s subsumes both $G^s_{e_1}$ and $G^s_{e_2}$, it subsumes them in the same set of stable instances. □

4.3 Using the Learned Definitions

The final goal of learning a definition for a target relation is to make predictions. In traditional relational learning, this is done efficiently by running the learned Horn definition over the underlying database [42, 32, 30, 36]. Performing prediction and inference is slightly different in our case as it must be done over a database that is *not* stable. Given a learned definition, one has to compute the subset of stable instances that contain the relations in the learned definition. This subset may be significantly smaller than the set of all stable solutions for the entire database. One may follow the semantic of certain answers and compute the final result of the definition as the intersection of its results on all stable instances [8, 9, 4]. Thus, the learned definition guides the user and reduces her effort needed for integration as the alternative approach is to compute all stable instances to learn an accurate definition. In particular, one may be able to learn an accurate definition using the relations of one database without any need to use the ones of other databases. The user can use those relations without any pre-processing effort for prediction. In this case, our approach helps users to avoid spending time and resources on cleaning and computing stable instances for learning and prediction. Users may also avoid computing stable instances by using methods to evaluate approximate answers to a query over the original and non-stable databases [8, 9]. Finally, one may also use the method proposed in Section 4.2 for computing coverage to determine whether the learned definition covers an input tuple by generating the ground bottom-clause for the input tuple and checking θ-subsumption for the produced ground bottom-clause and the learned definition. One can also apply the latter to determine the coverage of the input tuple if the learned definition is *not* stable.

5. COMMUTATIVITY OF LEARNING AND COMPUTING STABLE INSTANCES

As mentioned in Section 3, one approach to learning over a heterogeneous database with MDs is to first generate all stable instances, and then learn over them to generate a set of stable definitions. We show that the set of stable definitions of the definition learned by HDLearn over the input heterogeneous database is exactly the set of stable definitions learned by HDLearn over the stable instances of the input database. This property indicates that HDLearn leverages all information available in the heterogeneous database and learns all interesting patterns in its stable instances. In other words, the superior efficiency of HDLearn over the baseline approach of generating and learning over all stable instances of a database does *not* make HDLearn less effective than the baseline. Moreover, the algorithm of HDLearn over non-heterogeneous, i.e., stable, databases is essentially

the same as existing state-of-the-art relational learning algorithms [32, 33]. Thus, HDLearn has the potential to be effective over heterogeneous data.

Let *Learn*(J, E) be the function that takes as input a stable instance J and training examples E and returns a stable definition. The *LearnClause*(J, U, E_-) function in Algorithm 1 contains two steps: 1) build the bottom-clause C, denoted by function *BottomClause*(J, e), and 2) generalize C to generate C^*, denoted by function *Generalize*(C, J, E). Let *Learn*$_s(\mathbf{J}, E) = \{$ *Learn*$(J, E)\ \forall J \in \mathbf{J}\ \}$ be the function that returns a set of stable definitions, one for each stable instance $J \in \mathbf{J}$. We define functions *BottomClause*$_s(\mathbf{J}, e)$ and *Generalize*$_s(\mathbf{C}, \mathbf{J}, E)$ similarly. Let *Learn*$_d(I, E, \Sigma)$ be the function that returns a possibly non-stable definition H. We define functions *BottomClause*$_d(I, e, \Sigma)$, and *Generalize*$_d(C, I, E, \Sigma)$ similarly. Let *StableDefinitions*(H) return the set of stable definitions of H.

LEMMA 5.1. *We have*

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

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

Let *StableDefinitions*$(C) = \{C'_1, \ldots, C'_m\}$. We remove the literals that are not head-connected in each clause in $\{C'_1, \ldots, C'_m\}$. Let $\{J^{C'_1}, \ldots, J^{C'_m}\}$ be the canonical database instances of $\{C'_1, \ldots, C'_m\}$ [1]. This set can also be generated by running *StableInstances*(I^C, Σ), where I^C is the canonical database instance of C.

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

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

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

LEMMA 5.2. *We have*

$$\textit{Generalize}_s(\mathbf{C}, \textit{StableInstances}(I, \Sigma), E) = \textit{StableDefinitions}(\textit{Generalize}_d(C, I, E, \Sigma))$$

PROOF. We show that *Generalize*$_d(C, I, E, \Sigma) = C^*$ is a compact representation of *Generalize*$_s(\mathbf{C}, \mathbf{J}, e) = \{C^*_1, \ldots, C^*_n\}$, i.e.,
StableDefinitions$(C^*) = \{C^*_1, \ldots, C^*_n\}$.

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

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

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

Now consider two stable instances generated in *StableInstances*(I, Σ): J_1, which contains tuples $R_1(a, < b, b' >)$, $R_2(< b, b' >, c)$, $R_2(b'', c)$; and J_2, which contains tuples $R_1(a, < b, b'' >)$, $R_2(b', c)$, $R_2(< b, b'' >, c)$. The bottom-clause C_1 over instance J_1 has the form

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

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

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

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

Now assume that we want to generalize C_1 to cover another training example e'. Let $G_{e'}$ be the ground bottom-clause for e' and $G'_{e'}$ be a stable clause of $G_{e'}$. The literals in C_1 that are blocking will depend on the content of the ground bottom-clause $G'_{e'}$. Assume that the sets of literals $\{L'_1, \ldots, L'_n\}$ and $\{L_1, \ldots, L_n\}$ are equal. We consider the following cases for the literals that are not equal. The same cases apply when we want to generalize any other clause generated from a stable instance, e.g., C_2.

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

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

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

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

Case 5: $G'_{e'}$ does not contain either $R_1(a, < b, b' >)$ nor $R_2(< b, b' >, c)$. In this case, both $R_1(a, < b, b' >)$ and

$R_2(< b, b' >, c)$ are blocking; hence they are removed. $G_{e'}$ does not contain literals $R_1(a, b)$, $R_2(b', c)$, $m_\phi(b, b')$. Hence, these literals are also blocking literals in C and are removed.

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

Thus, we have the following theorem.

THEOREM 5.3. *The set of Horn definitions learned over each stable instance separately is exactly the set of stable Horn definitions obtained by learning over their original database instance, i.e.,*

$$Learn_s(StableInstances(I, \Sigma), E) = StableDefinitions(Learn_d(I, E, \Sigma))$$

Theorem 5.3 provides another interesting insight to the properties of the definition learned by HDLearn. If the stable instances of a database do *not* have sufficiently many tuples, HDLearn may learn a definition that has too many stable clauses without any (normal) literal in common according to Theorem 5.3. Such a definition may be hard to use and understand. In the worst case, if the stable instances do *not* have any tuple in common, HDLearn may learn a separate definition for each stable instance. Nevertheless, most tuples in the original database are usually shared between the stable instances as MDs do *not* normally apply to all attributes of all tuples. Furthermore, the user can always apply some of the matching literals in the learned definition to create a more general one that is easier to understand.

6. IMPLEMENTATION DETAILS

6.1 Efficient Learning Through Sampling

The set of tuples I_e created in bottom-clause construction may be large if many tuples in I are relevant to e. Thus, bottom-clause C_e would be very large, making the learning process extremely time-consuming. This problem is exacerbated when using similarity search operators to find relevant tuples, as many entities have similar names. To overcome this problem, HDLearn samples from the tuples in I_e to obtain a smaller tuple set $I_e^s \subseteq I_e$. To do so, HDLearn restricts the number of literals added to the bottom-clause per relation through a parameter called *sample size*. Let C_e be a bottom-clause associated with example e. A *sampled clause* 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 (lines 10-11 in Algorithm 2). We obtain a *uniform* sample I_R^s of I_R and only add tuples in I_R^s to I_e. In a uniform sample, every tuple in I_R is sampled independently with the same probability. This sampling technique is used by existing relational learning algorithms [32, 33].

6.1.1 Approximate Coverage Testing

As mentioned in Section 4.2, HDLearn uses θ-subsumption to compute the coverage of candidate clauses. Ideally, a ground bottom-clause G_e for example e must contain one literal per each tuple in the database that is connected to e through some (similarity) joins. Otherwise, the θ-subsumption test may declare that C does *not* cover e when C actually covers e. However, it is expensive to check θ-subsumption for large clauses. Therefore, to improve efficiency, we use sampling to build ground bottom-clauses. Because we use a sampled bottom-clause G_e^s, checking whether $C \subseteq_\theta G_e^s$ is an approximation of checking whether $I \wedge C \models e$. Our empirical results in Section 7.4 indicate that this issue does *not* significantly impact the accuracy of the learned definitions. The learning algorithm involves many (thousands) coverage tests. Because HDLearn reuses ground bottom-clauses, it can run efficiently over large databases.

6.2 Implementation of Matching Dependencies

HDLearn uses the similarity function defined as the average of the *Smith-Waterman-Gotoh* and the *Length* similarity functions. The *Smith-Waterman-Gotoh* function [22] measures the similarity of two strings based on their local sequence alignments. The *Length* function computes the similarity of the length of two strings by dividing the length of the smaller string by the length of the larger string. Given a string s_1, HDLearn considers a string s_2 similar to s_1 if their similarity score is greater than or equal to 0.65 and s_2 is within the top-k_m similar strings to s_1. In our experiments, we vary the value of k_m. To improve efficiency, we precompute the pairs of similar values according to the similarity function.

7. EXPERIMENTS

We empirically evaluate HDLearn to answer the following questions:

1. Can HDLearn learn over heterogeneous databases effectively and efficiently? (Section 7.2)
2. What is the benefit of using matching dependencies during learning? (Section 7.2)
3. How does the number of training examples affect HDLearn's effectiveness and efficiency? (Section 7.3)
4. How does sampling affect HDLearn's effectiveness and efficiency? (Section 7.4)

7.1 Experimental Settings

Systems:

1. **Baseline-1:** We use the relational learning system Castor [33] to learn over the original databases. Castor does *not* use MDs. We use Castor because it scales to large databases.
2. **Baseline-2:** We use Castor [33], but allow the attributes that appear in an MD to be joined through an exact join. Therefore, this baseline does use information from MDs but only considers exact matches between values.
3. **HDLearn:** We use HDLearn, as described in Section 4.

HDLearn uses the parameter k_b to restrict the sample size in (ground) bottom-clause construction. We fix k_b to 10. In Section 7.4, we evaluate the impact of this parameter on HDLearn's effectiveness and efficiency.

Datasets: We use three pairs of databases whose statistics are shown in Table 3.

1. **IMDB + OMDB**: The IMDB and OMDB databases contain information about movies [11]. We learn the target relation *dramaRestrictedMovies(imdbId)*, which contains the *imdbId* of movies that are of drama genre and are rated R. The *imdbId* is only contained in the IMDB database, the genre information is contained in both databases, and the rating information is only contained in the OMDB database. Therefore, in order to learn a definition for this

Name	#R	#T	#P	#N
IMDB	9	3.3M	100	200
OMDB	15	4.8M		
DBLP	4	15K	500	1000
Google Scholar	4	328K		
JMDB	43	9.2M	1000	2000
BOM	8	92K		

Table 3: Numbers of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset.

relation, one needs information from both databases. We specify the MD

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

The original database [11] contains ground truth that matches IMDB and OMDB movies. We use these to generate positive examples. To generate negative examples, we generate movies that are not of drama genre or rated R and sample them to obtain twice the number of positive examples. We refer to this dataset with one MD as **IMDB + OMDB (one MD)**. We also create MDs that match cast members and writer names between the two databases. We refer to the dataset that contains the three MDs as **IMDB + OMDB (three MDs)**.

2. **DBLP + Google Scholar**: The DBLP and Google Scholar databases contain information about academic papers [11]. We learn the target relation *gsPaperYear(gsId, year)*, which contains the Google Scholar id *gsId* and the *year* of publication of the paper as indicated in the DBLP database. We specify the MDs

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

The original database contains ground truth that matches DBLP and Google Scholar papers. We use these to generate positive examples. To generate negative examples, we generate pairs of Google Scholar ids and years of publication that are not correct, and sample them to obtain twice the number of positive examples.

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

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

Notice that the learning time of relational learning algorithms is not only affected by the number of training examples, but also by the number of tuples in the underlying database. HDLearn builds bottom-clauses for training examples. Learning over a large database often results in large bottom-clauses, which translates to long learning times. Therefore, learning over relational databases is generally time-consuming [33, 42]. In Section 7.3 we evaluate the impact of the number of training examples on HDLearn.

We perform 5-fold cross validation for all databases. We evaluate the F1-score and running time, showing the average over the cross validation. F1-score, which is the harmonic average of the precision and recall, measures the effectiveness of the learned definitions.

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

7.2 Learning Over Heterogeneous Databases

Table 4 shows the results of learning over all datasets using HDLearn and the baseline systems. Over all datasets, HDLearn obtains a better F1-score than the baselines. This is expected as HDLearn uses information in the input MDs to find relevant information from all databases. Baseline-1 does not learn any definition in the DBLP + Google Scholar dataset. This is because Castor (the system used in Baseline-1 (no MDs)) cannot access information from the DBLP database. Therefore, it is not able to find a reasonable definition. We see a similar result in the JMDB + BOM dataset. Baseline-2 is able to learn a definition over all datasets. However, as it relies on exact matches, the learned definitions are not as effective as the definitions learned by HDLearn. In particular, we see that Baseline-2 obtains a low F1-score over the JMDB + BOM dataset. This is because the same entities over these databases are consistently represented by different names. On the other hand, Baseline-2 obtains a good F1-score in the IMDB + BOM (three MDs) dataset. The MDs that match cast members and writer names between the two databases contain many exact matches. Therefore, the algorithm is able to find paths that connect movies in IMDB with movies in OMDB.

HDLearn is able to learn effective definitions over heterogeneous databases efficiently. In general, as k_m, the number of matches considered in MDs, increases, HDLearn's effectiveness increases. This shows that, even though the number of incorrect matches by the similarity function may increase, HDLearn is able to ignore these false matches during learning. In other words, using the training data, HDLearn figures that some patterns are useful and some are not. However, if HDLearn is learning a difficult concept, incorrect matches represent noise that may affect HDLearn's effectiveness. This is reflected when learning the *highGrossing* concept over the JMDB + BOM dataset, where the F1-score decreases when $k_m = 5$ and $k_m = 10$.

The extreme solution to this problem is to keep only one match. For instance, we may keep only the top similar match. This is actually what HDLearn with $k_m = 1$ does. This can be seen as generating only one stable instance and learning over it. Although efficient, the results are not always effective. Therefore, there is a benefit of looking at multiple stable instances for learning. As k_m increases, the learning time also increases. Therefore, there is a trade-off between the looseness of the similarity function used in MDs and HDLearn's effectiveness and efficiency.

The other extreme is to generate all stable instances, and then learn over each of them. However, there may exist a large number of stable instances. For instance, the movie

Dataset	Metric	Baseline-1 (no MDs)	Baseline-2 (exact match)	HDLearn			
				$k_m = 1$	$k_m = 2$	$k_m = 5$	$k_m = 10$
IMDB + OMDB (one MD)	F1-score	0.47	0.59	0.86	0.90	0.92	0.92
	Time (m)	0.12	0.13	0.18	0.26	0.42	0.87
IMDB + OMDB (three MDs)	F1-score	0.47	0.82	0.86	0.90	0.93	0.89
	Time (m)	0.12	0.48	0.21	0.30	25.87	285.39
DBLP + Google Scholar	F1-score	0	0.54	0.61	0.67	0.71	0.82
	Time (m)	2.5	2.5	3.1	2.7	2.7	2.7
JMDB + Box Office Mojo	F1-score	0	0.01	0.83	0.85	0.66	0.67
	Time (m)	0.3	0.8	5.2	8.5	24.2	33.7

Table 4: Results of learning over all datasets. Number of top similar matches denoted by k_m. Time refers to learning time in minutes.

American Splendor - 2003 matches 145 movies in the OMDB database. Just from this movie, we would get 145 distinct stable instances. Integrating and cleaning databases takes significant amount of time and manual labor. Materializing all stable instances would take a huge amount of space. Further, learning over each stable instance would be time-consuming. Therefore, the biggest benefit of HDLearn is that it avoids these problems by integrating "on-the-go".

7.3 Scalability of HDLearn

One advantage of relational learning algorithms is that they are data-efficient, i.e., they can learn effective definitions from a small number of training examples [16]. However, for a given learning task, it is not clear how many examples are needed. In this section, we evaluate the impact of the number of training examples in both HDLearn's effectiveness and efficiency. We use the IMDB + BOM (three MDs) dataset and fix $k_m = 2$. We generate 2100 positive and 4200 negative examples. From these sets, we use 100 positive and 200 negative examples for testing. From the remaining examples, we generate training sets containing 100, 500, 1000, and 2000 positive examples, and double the amount of negative examples. For each training set, we use HDLearn to learn a definition. Figure 1 (left) shows the F1-scores and learning times for each training set. With 100 positive and 200 negative examples, HDLearn obtains an F1-score of 0.80. With 500 positive and 1000 negative examples, the F1-score increases to 0.91. More training examples do not impact the F1-score significantly. On the other hand, the learning time consistently increases with the number of training examples. Nevertheless, HDLearn is able to learn efficiently even with the largest training set.

7.4 Impact of Sampling

In this section, we evaluate the impact of sampling (Section 6.1) on HDLearn's effectiveness and efficiency. We use the IMDB + BOM (three MDs) dataset and fix $k_m = 2$ and $k_m = 5$. We use 800 positive and 1600 negative examples for training, and 200 positive and 400 negative examples for testing. Figure 1 (middle and right) show the F1-score and learning time of HDLearn with $k_m = 2$ and $k_m = 5$, respectively. For both values of k_m, the F1-score does not change significantly with different sampling sizes. With $k_m = 2$, the learning time remains almost the same with different sampling sizes. However, with $k_m = 5$, the learning time increases significantly. Therefore, using a small sample size is enough for learning an effective definition efficiently.

8. RELATED WORK

Surveys on relational learning algorithms and applications can be found in [31, 21]. We have witnessed a surge in building efficient relational learning algorithms for large databases [42, 41, 18, 33]. Our work complements this line of work by creating an efficient relational learning system over heterogeneous databases.

Entity resolution is an active and important area of research in data management whose aim is to find whether two data values or tuples refer to the same real-world entity [20]. One may use these effort in implementing precise similarity operation in MDs.

There have been multiple efforts on modeling value and entity heterogeneity in databases using declarative constraints [17, 8, 5, 9]. Fan et al.have proposed algorithms to reason efficiently about combinations of MDs and other database constraints [17]. Some effort has been made to define the properties of a *good* clean and stable instance from all possible ones, e.g., based on the amount of modification made on the original database [8, 9]. However, in these cases one may end up with a large number of databases. Some researchers have defined the semantics of and algorithms for answering queries directly over the original database [8, 5]. Our effort extends the same approach of dealing with heterogeneous data by efficiently learning over the original database.

A different approach to querying and analyzing multiple databases is to use the available constraints, including MDs, define concrete similarity and matching functions for the databases, and create and materialize a unified database instance [10, 35, 13]. One may represent the inherent uncertainties in similarities between producing matched values as probabilities and store the final integrated database in a probabilistic database [38]. This approach is preferable if one is able to define a similarity function that accurately quantifies the degree of similarity between values in the domain and a matching function that is able to generate the correct unified value from its input. Such a matching function is hard and resource-intensive to define and usually requires a great deal of expert attention and manual labor. Also, one is *not* usually able to define a general one that can be used across various domains. Our goal is to create an learning algorithm that can be used over databases from various domains. Moreover, if the input databases are relatively large, the materialized database has to accommodate tuples from numerous stable instance causing the database to become extremely large. This in turn makes efficient learning over such a database challenging. Our proposed

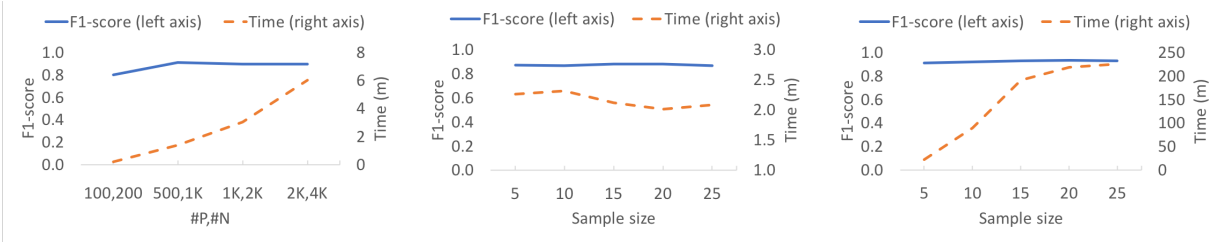

Figure 1: Results of learning over the IMDB+OMDB (three MDs) dataset while increasing the number of positive and negative (#P,#N) training examples (left) and while increasing sample size for $k_m = 2$ (middle) and $k_m = 5$ (right).

method is able to learn over the original database, which contains significantly fewer tuples, therefore, it avoids such a challenge.

ActiveClean gradually cleans a dirty dataset to learn a convex-loss model, such as Logistic Regression [28]. Its goal is to clean the underlying dataset such that the learned model becomes more effective as it receives more cleaned records. Cleaning may be done manually by an expert. Our objective, however, is to learn a model over dirty data without cleaning and transforming it. Furthermore, ActiveClean does *not* address the problem of having multiple cleaned versions of the database and assumes that the dataset has a unique clean instance.

9. REFERENCES

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

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

[3] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *2008 IEEE 24th International Conference on Data Engineering*, pages 40–49, April 2008.

[4] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[5] Z. Bahmani, L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Declarative entity resolution via matching dependencies and answer set programs. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*, 2012.

[6] Z. Bahmani, L. E. Bertossi, and N. Vasiloglou. Erblox: Combining matching dependencies with machine learning for entity resolution. In *SUM*, 2015.

[7] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: A generic approach to entity resolution. *The VLDB Journal*, 18(1):255–276, Jan. 2009.

[8] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory of Computing Systems*, 52:441–482, 2011.

[9] D. Burdick, R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. A declarative framework for linking entities. *ACM Trans. Database Syst.*, 41(3):17:1–17:38, July 2016.

[10] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2201–2206, 2016.

[11] S. Das, A. Doan, P. S. G. C., C. Gokhale, and P. Konda. The magellan data repository. `https://sites.google.com/site/anhaidgroup/projects/data`.

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

[13] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[14] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[15] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, Sept. 1997.

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

[17] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *Proc. VLDB Endow.*, 2(1):407–418, Aug. 2009.

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

[19] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 371–380, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[20] L. Getoor and A. Machanavajjhala. Entity resolution for big data. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1527–1527, New York, NY, USA, 2013. ACM.

[21] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[22] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162 3:705–8, 1982.

[23] A. Halevy, M. Franklin, and D. Maier. Principles of dataspace systems. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 1–9, New York, NY, USA, 2006. ACM.

[24] J. M. Hellerstein. People, computers, and the hot mess of real data. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 7–7, New York, NY, USA, 2016. ACM.

[25] M. Hernández, G. Koutrika, R. Krishnamurthy, L. Popa, and R. Wisnesky. Hil: A high-level scripting language for entity integration. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 549–560, New York, NY, USA, 2013. ACM.

[26] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 127–138, New York, NY, USA, 1995. ACM.

[27] D. V. Kalashnikov, L. V. Lakshmanan, and D. Srivastava. Fastqre: Fast query reverse engineering. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 337–350, New York, NY, USA, 2018. ACM.

[28] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. 9(12), 2016.

[29] H. Li, C.-Y. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13), 2015.

[30] L. Mihalkova and R. Mooney. Bottom-Up Learning of Markov Logic Network Structure. In *ICML*, 2007.

[31] S. Muggleton, L. Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan. Ilp turns 20. *Mach. Learn.*, 86(1):3–23, Jan. 2012.

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

[33] J. Picado, A. Termehchy, A. Fern, and P. Ataei. Schema Independent Relational Learning. In *SIGMOD*, 2017.

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

[35] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, Aug. 2017.

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

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

[38] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[39] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. Industry-scale duplicate detection. *Proc. VLDB Endow.*, 1(2):1253–1264, Aug. 2008.

[40] Z. Yan, N. Zheng, Z. G. Ives, P. P. Talukdar, and C. Yu. Actively soliciting feedback for query answers in keyword search-based data integration. *PVLDB*, 6(3):205–216, 2013.

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

[42] Q. Zeng, J. M. Patel, and D. Page. Quickfoil: Scalable inductive logic programming. *Proc. VLDB Endow.*, 8(3):197–208, Nov. 2014.