# Schema Independent and Scalable Relational Learning By Castor

Jose Picado  Parisa Ataei  Arash Termehchy  Alan Fern
School of EECS, Oregon State University, Corvallis, OR 97331
{picadolj,ataeip,termehca,afern}@oregonstate.edu

## ABSTRACT

Learning novel relations from relational databases is an important problem with many applications in database systems and machine learning. Relational learning algorithms leverage the properties of the database schema to find the definition of the target relation in terms of the existing relations in the database. However, the same data set may be represented under different schemas for various reasons, such as efficiency and data quality. Unfortunately, current relational learning algorithms tend to vary quite substantially over the choice of schema, which complicates their off-the-shelf application. We demonstrate *Castor*, a relational learning system that efficiently learns the same definitions over common schema variations. The results of Castor are more accurate than well-known learning systems over large data.

## 1. INTRODUCTION

Over the last decade, users' information needs over relational databases have expanded from answering precise queries to using machine learning in order to discover interesting and novel relations and concepts [5]. For instance, Table 1 shows fragments of the *original schema* for the UW-CSE[1] database. UW-CSE is a common relational learning benchmark that contains information about students, professors, courses, and publications. Given some examples of known student-advisor pairs, we may want to learn a new relation $advisedBy(stud, prof)$, which indicates that student $stud$ is advised by professor $prof$, according to available relations in the database. Machine learning algorithms often require to hand-engineer a set of features that capture the essential information required to predict the $advisedBy$ relation, where each feature is the result of a query to the database. We would then compute these features for each example in the training data, store the resulting feature vectors, and run a learning algorithm to learn the relation.

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

[1]http://alchemy.cs.washington.edu/data/uw-cse

*Proceedings of the VLDB Endowment,* Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

As opposed to "feature-based approaches", relational machine learning (also called relational learning or inductive logic programming) attempts to learn concepts directly from a relational database, without requiring the intermediate step of feature engineering [5, 4]. Given a database and training instances of a new target relation, relational learning algorithms attempt to induce (approximate) relational definitions of the target in terms of existing relations. Learned definitions are usually first-order formulas, often restricted to Datalog programs, which are easier to understand by users than the output of typical non-relational learning algorithms.

Because the space of possible definitions is enormous, relational learning algorithms must employ heuristics to search for accurate definitions. Unfortunately, such heuristics typically depend on the precise choice of schema for the underlying database, which means that the learning output is *schema dependent*. As an example, Table 1 shows parts of two schemas for the UW-CSE database. The *original schema* was designed by relational learning experts and is generally discouraged in the database community as it delivers poor usability and performance in query processing without providing any advantages in terms of data quality [1]. A database designer may use a schema closer to the *alternative schema* in Table 1, which is in 4th normal form. This would result in a more understandable schema and shorter query execution times, without introducing any redundancy. Note that restructuring the UW-CSE database from the original to alternative schema does not modify the content of the database; it only changes its organization. Let us use the classic relational learning algorithm FOIL [4] to induce a definition of $advisedBy(stud, prof)$ for both schemas of the UW-CSE database in Table 1. FOIL learns the following definition over the original schema on Table 1:

$$advisedBy(A, B) \leftarrow yearsInProgram(A, 7), publication(D, A), publication(D, B).$$

which covers 5 positive examples and 0 negative examples. On the other hand, FOIL learns the following definition over the alternative schema:

$$advisedBy(A, B) \leftarrow student(A, post\_generals, 5), professor(B, faculty), publication(C, B), taughtBy(D, B, E).$$

which covers 12 positive examples and 10 negative examples. Intuitively, the definition learned over the original schema better expresses the relationship between an advisor and advisee.

| Original Schema | Alternative Schema |
|---|---|
| student(<u>stud</u>) | student(<u>stud</u>,phase,years) |
| inPhase(<u>stud</u>,phase) | professor(<u>prof</u>,position) |
| yearsInProgram(<u>stud</u>,years) | publication(<u>title</u>,<u>person</u>) |
| professor(<u>prof</u>) | taughtBy(<u>crs</u>,<u>prof</u>,<u>term</u>) |
| hasPosition(<u>prof</u>,position) | |
| publication(<u>title</u>,<u>person</u>) | |
| taughtBy(<u>crs</u>,<u>prof</u>,<u>term</u>) | |

Table 1: Fragments of some schemas for UW-CSE data set. Primary key attributes are underlined.

Generally, there is no canonical schema for a particular set of content in practice. Therefore, people often represent the same information using different schemas [1]. For example, it is generally easier to enforce integrity constraints over highly normalized schemas [1]. On the other hand, because more normalized schemas usually contain many relations, they are hard to understand and maintain. It also takes a relatively long time to answer queries over database instances with such schemas [1]. Thus, a database designer may sacrifice data quality and choose a more *denormalized* schema to achieve better usability and/or performance. She may also hit a middle ground by choosing a style of design for some relations and another style for other relations in the schema. Moreover, the priorities of these objectives change over time.

Currently, users have to often restructure their databases to effectively use relational learning algorithms, i.e., deliver definitions for the target concepts that a domain expert would judge as correct. To make matters worse, these algorithms do not normally offer any clear description of their desired schema and users have to rely on their own expertise and/or do trial and error to find such schemas. Nevertheless, we ideally want our database analytics algorithms to be used by ordinary users, not just experts who know the internals of these algorithms. Further, the structure of large-scale databases constantly evolves and we want to move away from the need for constant expert attention to keep learning effective. One approach to solving this problem is to run a learning algorithm over all possible schemas and select the schema with the most accurate answers. However, computing all possible schemas of a database is generally undecidable. Even if one limits the search space to a particular family of schemas, the number of possible schemas is very large [1].

To make the learning algorithms more usable, we should move beyond the relational learning algorithms that are effective only over certain schemas for the data. We demonstrate *Castor*, a relational learning system that learns the same definition for the same training and input data regardless of the choices of schema. To achieve schema independence, it extends the traditional relational learning methods by using data dependencies, e.g. inclusion dependencies [1], in the database. Furthermore, as opposed to typical relational learning systems, it is implemented on top of an RDBMS, VoltDB (*voltdb.com*), which allows it to access data dependencies and scale for large databases. In particular,

- We show frequent types of schematic heterogeneity that are observed in real-world databases and show that current relational learning algorithms learn different definitions for the same data over these choices of schema.
- We demonstrate that Castor efficiently learns equivalent definitions for the same input data over a wide range of choices for the schema. We show that the results of Castor are more accurate or as accurate as other well-known learning systems.

## 2. FRAMEWORK

**Relational Learning:** An *atom* is a formula in the form of $R(u_1, \dots, u_n)$, where $R$ is a relation symbol. 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. Horn clauses are also called conjunctive queries. A *Datalog definition*, i.e., union of conjunctive queries, is a set of Horn clauses with the same positive literal. Relational learning algorithms learn Datalog definitions from input relational databases and training data. The learned definitions are called the hypothesis, which is usually restricted to non-recursive Datalog definitions without negation for efficiency reasons. Relational learning can be viewed as a search problem for a hypothesis that deduces the training data.

**Decomposition/ Composition:** *Decomposition* projects a relation to multiple relations. For example, the transformation from the alternative to original schema in Table 1 decomposes relation $student$ to relations $student$, $inPhase$, and $yearsInProgram$. Decomposition is used in several frequently applied schema normalizations, e.g., 3rd normal form. A decomposition preserves the content of a relation if the original and decomposed relations satisfy certain dependencies, e.g., functional or multivalued dependencies [1]. For example, in Table 1 attribute $stud$ functionally determines $phase$ and $years$ in both original and alternative schemas. These dependencies guarantee that no data item or tuple will be lost during the decomposition and that the original and decomposed databases contain the exact same information. Further, there should be inclusion dependencies, i.e., referential integrity constraints, between the common attributes in the decomposed relations. For instance, in the original schema of Table 1, there are inclusion dependencies between the attributes $stud$ in relations $inPhase$ and $yeasInProgram$ and attribute $stud$ in relation $student$. *Composition* joins multiple relations into a single relation. It is the inverse of decomposition. For example, the transformation from original to alternative schema in Table 1 composes relations $student$, $inPhase$, and $yearsInProgram$ into relation $student$. Schema denormalization is an example of composition. During the lifetime of a schema, one may decompose some relations and compose some other relations in the schema. We define a *decomposition/ composition* as a finite set of applications of composition or decomposition to a schema.

**Schema Independence:** A learning algorithm is *schema independent* if it learns semantically equivalent definitions over content-preserving transformations of the input database. For instance, for FOIL to be schema independent, it should learn the following definition over the alternative schema of Table 1 for the *adviseBy* relation as explained in Section 1.

$$advisedBy(A,B) \leftarrow student(A,C,7), publication(D,A),$$
$$publication(D,B).$$

This definition is semantically equivalent to the one learned by FOIL over the original schema, which is given in Section 1. Schema independence can be defined on various types of content-preserving schema transformations. In this demonstration, we focus on schema independence over decomposition/ composition. The reasons for selecting decomposition/ composition are twofold. First, they are used in most normalizations and de-normalizations, which are arguably one of the most frequent schema modifications [1]. We also observe several cases of them in relational learning benchmarks, one of which is shown in Table 1.

## 3. CASTOR ALGORITHM

As many relational learning algorithms, Castor follows a covering approach. Algorithm 1 depicts Castor's learning algorithm. It constructs one clause at a time. If the clause satisfies the minimum criterion, Castor adds the clause to the learned definition and discards the positive examples covered by the clause. It stops when all positive examples are covered by the learned definition. Castor

**Algorithm 1:** Castor's cover-set algorithm.

```
Input   : Database instance $I$, positive examples $E^+$, negative
          examples $E^-$, sample size $K$, beam width $N$
Output: A set of Horn definitions $H$
$H = \{\}$;
$U = E^+$;
while $U$ is not empty do
    $C = LearnClause(I, U, E^-, K, N)$;
    if $C$ satisfies minimum criterion then
        $C' = Reduce(C)$;
        $H = H \cup C'$;
        $U = U - \{c \in U | H \wedge I \models c\}$;
return $H$ ;
```

accepts a clause only if its precision and F1-score, i.e., harmonic average of precision and recall, are greater than those of a random classifier. Other algorithms check only for the precision of learned clauses [2]. These algorithms usually overfit to the training data, as they learn clauses that contain too many constants. This happens more often for schemas whose relations have relatively many attributes, e.g., the alternative schema in Table 1. For instance, if an algorithm checks only for precision, it learns the following clause over the alternative schema in Table 1:

$$advisedBy(A, B) \leftarrow student(A, post\_generals, 5), \; publication(C, A), publication(C, B).$$

On the other hand, Castor, which checks for both precision and F1-score, learns more general clauses, such as

$$advisedBy(A, B) \leftarrow student(A, D, E), \; publication(C, A), publication(C, B).$$

An algorithm that only checks for precision may not overfit over schemas whose relations have relatively small number of attributes, e.g., the original schema in Table 1. For instance, it may learn the following clause over the original schema in Table 1.

$$advisedBy(A, B) \leftarrow student(A), \; publication(C, A), publication(C, B).$$

This means that such algorithm may return different answers over the original and alternative schemas and is not schema independent. However, because Castor checks for both precision and F1-score, it does not suffer from this problem and learns accurate and general clauses over both schemas. Castor follows the bottom-up method for relational learning [2]. First, it constructs the most specific clause that covers a given positive example, relative to the database instance, called *bottom clause*. Then, it *generalizes* the bottom clause to cover as most positive and as fewest negative examples as possible.

## 3.1 Bottom Clause Construction

To compute the bottom clause associated with example $e$ and relative to database $I$, Castor assigns fresh variables to constants in $e$ and maintains this mapping in a hash table. It creates the head of the bottom clause by replacing the constants in $e$ with their assigned variables. The algorithm selects all tuples in the database that contain at least one constant in the hash table. For each tuple, it creates a new literal with the same relation name as the tuple and adds the literal to the body of the bottom clause. If the literal (tuple) has a new constant, the algorithm assigns a fresh variable to the constant and adds the new mapping to the hash table. In each following iteration, the algorithm selects tuples in the database that contain the newly added constants to the hash table and adds their corresponding literals to the clause. This procedure generates very long clauses over a large database after a small number of iterations, which takes a very long time to construct and then generalize. A common method is to restrict the maximum number of literals, called $clauseLength$, in the body of the bottom clause [2].

For example, assume that we would like to learn relation $hardWorking(x)$, which indicates that a student is hardworking, over the UW-CSE database. Let $clauseLength$ be 2. The bottom clause construction algorithm starts with a positive example , e.g., $hardWorking(Mary)$, assigns a fresh variable $v_1$ to $Mary$, and adds the literal $hardWorking(v_1)$ to the head of the bottom clause. It then finds all tuples that contain constant $Mary$. Assume that $Mary$ appears only in the $student$ and $inPhase$ relations in the original schema and only in the $student$ relation in the alternative schema introduced in Table 1. Hence, the bottom clause construction adds the corresponding literals to the bottom clause and stops when there are two literals in the bottom clause. Over the original schema, the bottom clause algorithm delivers the clause $hardWorking(v_1) \leftarrow student(v_1), inPhase(v_1, v_2)$. On the other hand, over the alternative schema it generates the clause $hardWorking(v_1) \leftarrow student(v_1, v_2, v_3)$.
These two clauses are not semantically equivalent. Hence, the bottom clause construction algorithm may deliver different results for the same example and equivalent instances of schemas representing the same information. More importantly, it may miss some important tuples over some schemas, e.g., $yearsInProgram(Mary, 2)$ over the original schema. To overcome this problem, Castor uses inclusion dependencies to construct bottom clauses. More precisely, after selecting a tuple, Castor applies inclusion dependencies to find other tuples related to the selected tuple and adds them to the bottom clause. For example, after it selects tuple $student(Mary)$ over the database with original schema in our example, it also selects tuples $inPhase(Mary, PostPrelims)$ and $year(Mary, 2)$ as they satisfy $inPhase[stud] \subseteq student[stud]$ and $yearsInProgram[stud] \subseteq student[stud]$, respectively.

## 3.2 Generalization

Castor first creates the bottom clause for a given positive example $e$. It then generalizes this bottom clause iteratively. Given clause $C$, Castor randomly picks a subset $E^+$ of positive examples to generalize $C$. For each example $e'$ in $E^+$, Castor generates a candidate clause $C'$, which is more general than $C$ and covers $e'$. It uses the $armg$ operator [2], which drops literals in the body of $C$ that do not cover $e'$. Castor then selects the highest scoring candidate clauses and iterates until the clauses cannot be improved. The $armg$ operator may generate non-equivalent clauses from semantically equivalent clauses over a database and its composition/decomposition. For example, given the bottom clauses over the original and alternative schemas in the example in Section 3.1, the literal $student(v_1)$ may satisfy example $e'$, but $inPhase(v_1, v_2)$ may not. Hence, the algorithm keeps $student(v_1)$ and removes $inPhase(v_1, v_2)$ from the bottom clause generated of the database over the original schema. Because both databases in the example have the same content, literal $student(v_1, v_2, v_3)$ will not satisfy $e'$ and will be removed by the algorithm from the bottom clause over the alternative schema. To solve this issue, immediately after removing a literal $L_1$ with relation symbol $R$, Castor also removes literal $L_2$ with relation symbol $S$ such that $R[X] \subseteq S[Y]$ is an inclusion dependency in the schema and $L_1$ and $L_2$ share the same variables and/or constants in attributes $X$ and $Y$. For example, Castor removes literal $student(v_1)$ after removing $inPhase(v_1, v_2)$ in the example due to the inclusion dependency $inPhase[stud] \subseteq student[stud]$ over the database with original schema. Clauses are further generalized by removing literals that are non-essential, shown as $Reduce()$ function in Algorithm 1. A literal is non-essential if, after removed from a clause, the number of negative examples covered by the clause does not increase [2]. Castor uses inclusion dependencies to generate equivalent clauses in this step.

## 4. CASTOR ARCHITECTURE

Castor performs several optimizations to improve the efficiency of the learning algorithm. First, Castor removes redundant literals in bottom clauses. A literal $L$ in clause $C$ is redundant if $C$ is equivalent to $C' = C - \{L\}$. Castor checks clause equivalence by using theta-transformation, which is an approximation of the clause equivalence problem that retains the property of correctness. Second, Castor optimizes the generalization process by reducing the number of coverage tests. If clause $C$ covers example $e$, then clause $C''$, which is more general than $C$, also covers $e$. If Castor knows that $C$ covers $e$, it does not check if $C''$ covers $e$. Third, the bottom clause construction algorithm is implemented inside a stored procedure. We implement Castor on top of the in-memory RDBMS VoltDB. Because stored procedures in VoltDB are pre-compiled, bottom clause construction is very efficient. Figure 1 sketches the high-level architecture of Castor. The first time that Castor is run on a schema, it creates the stored procedure that implements the bottom clause construction algorithm for the given schema. Castor reuses the stored procedure when the algorithm is run again, with either new training data or updated database instance.

Table 2 shows the results of running Castor and two other relational learning algorithms over a subset of the HIV database[2]. We learn the relation $hiv\_active(compound)$, which indicates that $compound$ has an anti-HIV activity. We use two schemas, which are a composition/ decomposition of each other. The database contains 14M tuples over schema 1 and 7.8M tuples over schema 2. We use the implementation of FOIL and Progol based on the popular relational learning library Aleph[3]. We cannot compare our system with QuickFOIL [5], as it is not available for download. We also evaluated ProGolem [2], however it did not terminate after 10 hours. All experiments were run on a 2.6GHz Intel Xeon E5-2640 processor, running CentOS Linux 7.2 with 50GB of main memory.

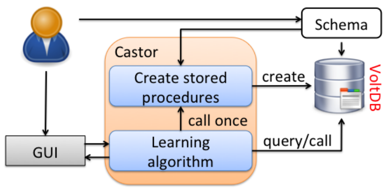


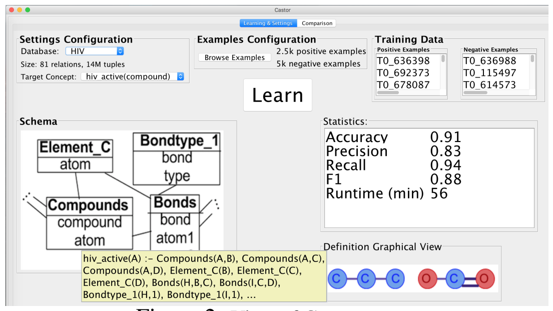
Figure 1: High-level architecture of Castor.

| Algorithm | Metric | Schema 1 | Schema 2 |
|---|---|---|---|
| FOIL | Precision | 0.68 | 0.66 |
| | Recall | 0.85 | 0.94 |
| | Time (min) | 9.4 | 3 |
| Progol | Precision | 0.71 | 0.64 |
| | Recall | 0.88 | 0.94 |
| | Time (min) | 36.8 | 16.3 |
| Castor | Precision | 0.83 | 0.83 |
| | Recall | 0.94 | 0.94 |
| | Time (min) | 56 | 20.7 |

Table 2: Results of different learning algorithms over HIV data.

## 5. DEMONSTRATION

In our demonstration, we first show examples of different styles of schema design using benchmark databases, such as UW-CSE, and real-world databases, such as HIV and IMDb. Users may select a database and training data, run Castor and observe the graphical view and Datalog of the learned definition, as well as the accuracy of the learned definition as shown in Figure 2. For each database, users will also see a visual representation of the database schema. Users can view a list of prepared composition/ decomposition transformations for a database, select their desired transformation, and observe the results of Castor on the original database and its composition/ decomposition. Users can also visually explore, decompose/ compose the schema of the database, run Castor, and observe the results of Castor on the original and transformed database. In addition to Castor, our prototype has also the implementations of well-known relational learning algorithms: FOIL, Progol, and ProGolem [4, 2]. Users can select one of these algorithms, a database and training data, see the learned definitions of Castor and the selected algorithm over the chosen database, and compare the accuracy and efficiency of the selected algorithms with Castor's. Further, they can select a composition/ decomposition for the chosen database and compare the robustness of the chosen algorithm with Castor as shown in Figure 3.



Settings Configuration
Database: HIV
Size: 81 relations, 14M tuples
Target Concept: hiv_active(compound)

Examples Configuration
2.5k positive examples
5k negative examples

Training Data

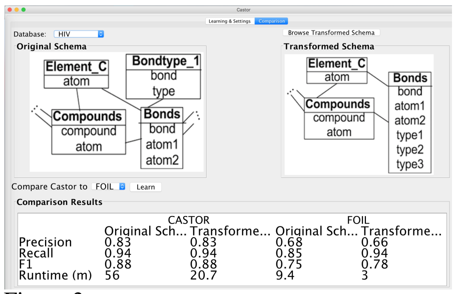
| Positive Examples | Negative Examples |
|---|---|
| T0_636398 | T0_636988 |
| T0_692373 | T0_115497 |
| T0_678087 | T0_614573 |

Statistics:

```
Accuracy        0.91
Precision       0.83
Recall          0.94
F1              0.88
Runtime (min)  56
```

hiv_active(A) :– Compounds(A,B), Compounds(A,C), Compounds(A,D), Element_C(B), Element_C(C), Element_C(D), Bonds(H,B,C), Bonds(I,C,D), Bondtype_1(H,1), Bondtype_1(I,1), ...

Figure 2: View of Castor output.



Database: HIV

Compare Castor to FOIL

Comparison Results

```
               CASTOR                       FOIL
               Original Sch... Transforme... Original Sch... Transforme...
Precision      0.83            0.83          0.68            0.66
Recall         0.94            0.94          0.85            0.94
F1             0.88            0.88          0.75            0.78
Runtime (m)    56              20.7          9.4             3
```

Figure 3: Comparison of Castor to other algorithms.

## 6. ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under grant IIS-1423238 and sponsored by a subcontract with Raytheon BBN Technologies Corp. under a prime contract with the Air Force Research Laboratory (AFRL).

## 7. REFERENCES

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

[2] S. Muggleton, J. C. A. Santos, and A. Tamaddoni-Nezhad. ProGolem: A System Based on Relative Minimal Generalisation. In *ILP*, volume 5989, 2009.

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

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

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

[2]https://wiki.nci.nih.gov/display/NCIDTPdata/

[3]www.cs.ox.ac.uk/activities/machlearn/Aleph/aleph.html