Inductive Database Design

Hendrik Blockeel and Luc De Raedt

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
email : {Hendrik.Blockeel,Luc.DeRaedt}@cs.kuleuven.ac.be

Abstract. When designing a (deductive) database, the designer has to decide for each predicate (or relation) whether it should be defined extensionally or intensionally, and what the definition should look like. An intelligent system is presented to assist the designer in this task. It starts from an example database in which all predicates are defined extensionally. It then tries to compact the database by transforming extensionally defined predicates into intensionally defined ones. The intelligent system employs techniques from the area of inductive logic programming.

Keywords: Logic for Artificial Intelligence, Learning and Knowledge Discovery, Database Design, Inductive Logic Programming

1 Introduction

When designing databases, the designer has to determine the structure of the database by determining the extensional and intensional predicates, and by providing definitions for each of the intensional predicates. At present, there exists few guidelines to help the designer in this task, which makes designing deductive databases a hard task. Nevertheless, the design ultimately determines the quality of the database.

In this paper, we present a novel approach, called inductive database design, to assist database designers in their task. The key assumption underlying inductive database design, is that it is often easy to give an example state of the database in which all predicates (or relations) are defined extensionally. Given an example state, techniques from the field of inductive logic programming can be used to discover clauses that are valid in the example state. The inductive database design method we propose employs these clauses in order to transform some of the extensional predicates into intensional ones. During this process it is guided by the principle of compaction, which states that the better database is the more compact one, i.e. the one that requires less memory. Though the technique we present can – in principle – be used fully automatically, we believe it is more adequate to view it as an intelligent assistant that gives advice to the database designer, who should be given the opportunity to reject the definitions proposed by the system.

The paper is organised as follows: in Section 2, we review some concepts of deductive databases and logic programming, in Section 3, we review the inductive logic programming system CLAUDIEN [1, 8], which will be adapted for use in our

inductive database design tool, in Section 4, we address the problem of finding intensional definitions for predicates, in Section 5, we present an experiment, and finally, in Section 6, we conclude and touch upon related work.

2 Logic Programming Concepts and Datalog

The database framework we will use is based on the Datalog subset of clausal logic. More specifically, we will use the follow notions.

A *clause* is a universally quantified logical formula of the form $A_1, ..., A_m \leftarrow B_1, ..., B_n$ where the A_i and B_j are positive literals (atoms). The above clause can be read as A_1 or ... or A_m if B_1 and ... and B_n. Extending the usual convention for *definite clauses* (where $m = 1$), we call $A_1, ..., A_m$ the *head* and $B_1, ..., B_n$ the *body* of the clause. A *fact* is a definite clause with empty body ($m = 1$, $n = 0$).

Definition 1. A clause is range-restricted iff all variables occurring in the head also occur in the body.

Range-restriction is often imposed in the database literature, it allows to avoid the derivation of non-ground true facts.

Definition 2. An extensional predicate is a predicate defined by a set of functor-free ground facts.

Definition 3. An intensional predicate is a predicate defined by a set of functor-free and range-restricted definite clauses.

Definition 4. An integrity constraint is a range-restricted functor-free clause.

Definition 5. A deductive database is a couple $\langle \mathcal{E}, \mathcal{I} \rangle$, where $\mathcal{E}$ is a set of extensional predicate definitions and $\mathcal{I}$ is a set of intensional predicate definitions.

Definition 6. A clause c is valid in a definite clause theory T iff c is true in the least Herbrand model of T.

To verify whether a clause $A_1, ..., A_m \leftarrow B_1, ..., B_n$ is true in a definite clause theory T, one can run the query $\leftarrow B_1, ..., B_n, \neg A_1, ..., \neg A_m$ on a database containing T. If the query finitely fails, the clause is valid, otherwise it is invalid.

3 Inductive Logic Programming

In this section, we give a brief overview of the inductive logic programming system CLAUDIEN [1, 8, 2, 3].

The CLAUDIEN system starts from a definite clause theory T and a language L (which is a set of well-formed clauses) and finds a set of maximally general clauses that are valid in the given theory.

Definition 7. Clause c_1 is more general than clause c_2 iff there exists a substitution θ such that $c_1\theta \subset c_2$ (where clauses are seen as sets of literals).

We will also say that c_1 θ-subsumes c_2 (cf. Plotkin [6]). As shown by Plotkin, θ-subsumption induces a partial order on the set of all possible clauses, which is exploited by many inductive logic programming systems [5].

We can now illustrate the CLAUDIEN setting. Given the database

$$\{male(maarten), female(soetkin), human(maarten), human(soetkin)\}$$

and as L the set of all constant-free clauses containing maximum 3 literals, CLAUDIEN would discover the following clauses, which are all valid in the database:

$$human(X) \leftarrow male(X)$$

$$human(X) \leftarrow female(X)$$

$$\leftarrow male(X), female(X)$$

$$female(X), male(X) \leftarrow human(X)$$

Roughly speaking, CLAUDIEN works as follows (cf. Figure 1). It keeps track of a list of candidate clauses Q, which is initialised to the maximally general clause (in L). It repeatedly deletes a clause c from Q, and tests whether c is valid in the theory. If it is, c is added to the final hypothesis, otherwise, all maximally general specialisations of c (in L) are computed (using a so-called *refinement operator* ρ) and added back to Q. This process continues until Q is empty and all relevant parts of the search-space have been considered. It should be mentioned that CLAUDIEN employs several techniques to prune and optimize the search, cf. [2].

$Q := \{max(L)\}$
$H := \emptyset$
while $Q \neq \emptyset$ **do**
  delete c from Q
  if c is valid in T
  then add c to H
  else add $\rho(c)$ to Q

Fig. 1. The CLAUDIEN algorithm (simplified)

To specify the set of clauses to consider, i.e. the language bias, CLAUDIEN employs so-called dlab_templates (cf. [2, 4]). For instance:

```
dlab_template('Q(X,Y) <-- 1-4:[P(X), P(Y), Q(X,Y), Q(Y,X)]').
dlab_var('P', 1-1, [male, female]).
dlab_var('Q', 1-1, [parent, father, mother]).
```

specifies that clauses should have one binary predicate in the head, and one to four predicates in the body. The dlab_variables P and Q are placeholders for real predicates; they can be substituted by the predicates in the corresponding `dlab_var` declaration. Several occurrences of a dlab_variable can be substituted by different predicates.

For instance, if unary predicates are `male` and `female`, and binary predicates are `parent`, `father` and `mother`, then

$$father(X,Y) \leftarrow male(X),\ parent(X,Y)$$

is part of the specified language.

4 Finding Intensional Definitions

As intensional definitions typically consist of a small set of rules, while extensional definitions consist of a large set of facts, a database will be much more compact if many predicates are defined intensionally. The question is then, how extensional definitions can be turned into intensional ones in such a way that the database becomes as compact as possible.

We will use the following approach. For each predicate, we try to find a set of clauses that intensionally defines the predicate. These clauses should be as simple as possible. They can be found using a modified version of CLAUDIEN. In a second step, we show how multiple predicates can be defined intensionally using the results of the first step.

4.1 Defining One Predicate Intensionally

While the CLAUDIEN system was designed to find all the clauses C in a hypothesis space that are valid in a database, our goal is only to find a small number of clauses which together form a good intensional definition. To this end, CLAUDIEN is run a number of times, and each time one best clause is chosen, "best" meaning that the clause predicts more facts that have not been predicted yet, than any other clause.

Because in each run, we are only interested in the best clause, it is possible to prune the search space much more than when all the solutions have to be found. Instead of only pruning branches that cannot lead to a solution, CLAUDIEN can now prune any branch that cannot lead to a *better* solution than the best up till now. This causes a great gain in efficiency, compared to the "standard" CLAUDIEN system.

The modified CLAUDIEN algorithm, which we call CLAUDIEN*, is shown in Figure 2.

CLAUDIEN* will only generate clauses within the language bias L that define the predicate p in terms of the predicates in P. The rootclauses $\{\max(L)\}$ with which Q is initialized, depend on the language bias that is used.

We define the compaction achieved by a clause c as $comp(c) = C(c) - P(c)$, where $C(c)$ is the complexity of the clause (the number of literals in it) and $P(c)$

Procedure Claudien*(p : predicate, P : set of predicates) returns clause
$Q := \{\max(L)\}$
$c_{\text{best}} := none$
while $Q \neq \emptyset$ **do**
delete c from Q
if c is valid in T **then** $c_{\text{best}} := c$
else
add $\rho(c)$ to Q
remove from Q *every* r *where* $comp(r) \geq comp(c_b)$
return c_{best}

Fig. 2. CLAUDIEN*, the modified CLAUDIEN algorithm

the number of facts that can be derived using the clause. $P(c)$ is computed as indicated in Figure 3. As a fact consists of one literal, $comp(c)$ is indeed the difference in the number of literals in the database, when the predicted facts are replaced by the clause.

Function $P(c : Clause)$ **returns integer**
Let p be the predicate in $head(c)$
Let DB be the database obtained by deleting all facts for p
Let H_p be the set of definite clauses already found for p
$P(c) = \mid covers(DB \wedge H_p \wedge c) \mid - \mid covers(DB \wedge H_p) \mid$

Fig. 3. Computing the number of new facts covered by a clause

The covers relation employed is intensional coverage, i.e. $covers(DB)$ is the set of all ground facts logically entailed by DB.

It is easy to prove that this search algorithm is admissible, i.e. that the best solution will always be found.

Theorem 8. *Let c and d be clauses and let b be the best clause found so far. If $comp(c) > comp(b)$ then for every refinement d of c, $comp(d) > comp(b)$.*

Proof. CLAUDIEN* refines clauses only by adding literals of the form $P(\ldots)$ to them, where P is a database predicate. This increases the complexity of the clause. Therefore, as d is a refinement of c, $C(d) > C(c)$. On the other hand, because $body(d) \supseteq body(c)$, the set of facts predicted by d must be a subset of the set of facts predicted by c, hence $P(d) \leq P(c)$.

In our implementation, the heuristic used to order the clauses in Q, is $C(c) + N(c) - P(c)$, where $N(c)$ is the number of facts covered by a clause that should

not be covered (so it gives some idea of how much the clause will still have to be refined before becoming valid). Although this heuristic does not influence the solution that will be found, a good heuristic will lead to better clauses being found earlier, and will therefore allow more pruning.

As CLAUDIEN*, only finds one clause (the most compacting one), and this clause is not necessarily complete as an intensional definition (there may still be facts that cannot be derived by it), it has to be run repeatedly until every fact of the predicate can be derived. This leads to the FID (Find an Intensional Definition) algorithm, which shown in Figure 4.

When computing the compaction caused by a clause, only facts that are predicted by the clause *and that are not predicted using any already existing rules* should be taken into account. That is why the facts predicted by a clause that is added to the definition are marked.

```
Function FID(p : predicate, P : set of predicates) returns
        intensional definition of p using predicates in P or false
  D := ∅
  while D is not complete do
    run CLAUDIEN*
    if a new clause c has been found
    then add c to D, and mark the facts predicted by c
    else return false
  if D is complete then return D
```

Fig. 4. FID: an algorithm for finding an intensional definition

If an intensional definition for a predicate p is found by the above algorithm, it is guaranteed to be *sound and complete*. Soundness means that no facts can be derived using this definition, that were not in the original database. Completeness means that every fact in the extensional definition of the predicate can be derived with the intensional definition. It is easy to see that these properties hold.

Theorem 9. *Any intensional definition found by FID is complete.*

Proof. This follows trivially from the fact that the algorithm only returns an intensional definition for p if all the facts for p are indeed predicted by it.

Theorem 10. *Any intensional definition found by FID is sound.*

Proof. The intensional definition consists of valid rules (as CLAUDIEN* does not induce invalid rules). Now let S be the set of facts that is implied by the original database. For each application of any individual rule in the intensional definition, if before the application of the rule no facts are in the database that are not in

$\mathcal{S}$, the application of the rule itself can only allow the derivation of facts that are in $\mathcal{S}$ (because the rule is valid). Repeated application of one or more rules for p will not change this condition.

The completeness and soundness properties of an intensional definition guarantee that by replacing an extensional definition with an intensional one, no information is lost nor gained: the new database is equivalent to the original one.

4.2 Finding Several Definitions

Having an algorithm that finds an intensional definition for one predicate, it may seem a trivial task to define as much predicates as possible intensionally. However, this is not so easy. The fact that two predicates may have an intensional definition does not imply that they can both be defined intensionally at the same time. An intensional definition of one predicate may preclude an intensional definition of another predicate. The following example illustrates this.

Example 1. Suppose we have a tiny database which contains the following extensional definitions:

```
p(1).  p(2).  p(3).          q(1).  q(2).  q(3).
```

There exists an intensional definition for p, which can replace its extensional definition:

```
p(X) :- q(X).          % intensional definition for p
```

and similarly, q can be defined intensionally using the following rule:

```
q(X) :- p(X).          % intensional definition for q
```

However, we cannot replace both extensional definitions by their corresponding intensional definition, as that would destroy the original information (the least Herbrand model would be empty).

This shows that not every predicate for which an intensional definition exists, will indeed be defined intensionally.

To compute which predicates to define intensionally, we use the algorithm in Figure 5. This algorithm computes a partition $\mathcal{E}, \mathcal{I}$ of the predicates $\mathcal{P}$. First, it initializes $\mathcal{E}$ to the set of predicates for which FID is unable to construct intensional definitions. For every predicate that is not in this initial set $\mathcal{E}$, an intensional definition exists, but whether this definition will be used to define the predicate depends on how the other predicates are defined. It is, however, possible to find a set $\mathcal{I}'$ of predicates that can certainly be defined intensionally, irrespectively of what is done with the other predicates.

Consider the predicates that can be defined using only predicates initially in $\mathcal{E}$. (The term "uses" is only used here with respect to *other* predicates than the

$\mathcal{P}$:= set of all predicates
for all $p \in \mathcal{P}$ **do**
    let $f(p) := FID(p, \mathcal{P})$
$\mathcal{E} := \{p \in \mathcal{P} \mid f(p) = false\}$
$\mathcal{I}' := \emptyset$
repeat
    repeat
        $\mathcal{I}' := \{p \in \mathcal{P} \mid f(p)$ contains only predicates in $\mathcal{E} \cup \mathcal{I}' \cup \{p\}\}$
    until $\mathcal{I}'$ does not change anymore
    if $\mathcal{I}'$ has changed in the above loop
    then
        $\mathcal{M} := (\mathcal{P} - \mathcal{I}') - \mathcal{E}$
        for all p in $\mathcal{M}$:
            if $FID(p, \mathcal{E} \cup \mathcal{I}' \cup \{p\}) \neq false$
            then $f(p) := FID(p, \mathcal{E} \cup \mathcal{I}' \cup \{p\})$
until $\mathcal{I}'$ does not change anymore
$\mathcal{I} := \mathcal{I}'$
repeat
    let p_b be the predicate in $\mathcal{M}$ for which $f(p_b)$ achieves maximal compaction
    add p_b to $\mathcal{I}$ and remove p_b from $\mathcal{M}$
    for all $p \in \mathcal{M}$ **do**
        if $f(p)$ is incorrect given $\mathcal{I}$ and $\mathcal{E}$
        then add p to $\mathcal{E}$
until $\mathcal{M} = \emptyset$
define the predicates p in $\mathcal{I}$ intensionally using $f(p)$

Fig. 5. Computing $\mathcal{E}$, $\mathcal{I}$

one being defined; the latter one is always allowed to occur in its own definition, so that direct recursion is allowed.) We call this set of predicates $\mathcal{I}'_1$. We successively define the sets $\mathcal{I}'_n$ as the sets of predicates that can be defined using only predicates in

$$\bigcup_{i<n} \mathcal{I}'_i \cup \mathcal{E}'$$

Finally, $\mathcal{I}'$ is defined as

$$\mathcal{I}' = \bigcup_{i=1}^{\infty} \mathcal{I}'_i$$

$\mathcal{I}'$ is computed in the first, inner repeat loop of Algorithm 5.

The set of predicates that are not in $\mathcal{E}$ initially nor in $\mathcal{I}'$ is called $\mathcal{M}$. In the final database $\mathcal{DB}$ all the predicates in $\mathcal{I}'$ will belong to $\mathcal{I}$, whereas some of the predicates in $\mathcal{M}$ will belong to $\mathcal{E}$ and some to $\mathcal{I}$.

Since for each predicate only one definition was returned by the above algorithm, it is possible that some predicates that are now in $\mathcal{M}$ in fact should belong to $\mathcal{I}$, i.e. some definition using only predicates initially in $\mathcal{E}$ and $\mathcal{I}'$ exists but was not returned by FID.

To make the final $\mathcal{I}$ set as large as possible, it is therefore useful to rerun the FID procedure with a reduced search space, now allowing only predicates in $\mathcal{E}$ and those predicates already known to belong to $\mathcal{I}$ in the body of the clauses. This can be done iteratively until no predicates are added to $\mathcal{I}$. This is realized in the outer repeat loop of the algorithm.

Of the predicates that remain in $\mathcal{M}$, the algorithm must then decide which predicates should be defined intensionally, and which extensionally. This is done using a hill-climbing approach (the last loop of Figure 5). In each step, the intensional definition that leads to the greatest compaction is chosen. This may make other intensional definitions invalid, so in the next step a smaller set of intensional definitions will be considered. This is continued until no new predicates can be defined intensionally.

5 An experiment

We have run our system on a small family database. This database contains extensional definitions for the predicates *parent*, *married*, *married1* (an asymmetrical relation containing only the couple (X, Y) when *married* contains both (X, Y) and (Y, X)), *grandparent*, *sibling*, *sibling-in-law*, *parent-in-law*, *grandparent-in-law*, *aunt-or-uncle* and *niece-or-cousin*. All together, the extensional definitions contain 723 facts.

In the first run, FID returns a definition for every predicate except *parent* and *married1*. Most of these definitions are found in a couple of minutes; the total time for all the predicates is about 40 minutes. As $\mathcal{E} = \{parent, married1\}$, the system first adds {*grandparent, parent-in-law, sibling, married*} and then {*sibling-in-law, aunt-or-uncle, grandparent-in-law*} to $\mathcal{I}$, so that finally $\mathcal{M} = \emptyset$, and no extra runs of FID are needed.

When the database is redesigned in this way, its size is reduced from 723 literals to 117 (of which 88 for the extensional definitions, and 29 for the intensional definitions). It is clear that the new database must have a close to minimal size, as the information about parenthood and marriage is crucial, and all the other predicates are derived from this using a simple definition.

6 Conclusions

In this paper, we have discussed the problem of designing a deductive database in such a way that it becomes simpler and more structured. We have presented a novel approach to this problem, called inductive database design, in the form of a system that automatically derives rules and builds intensional definitions, using the compaction caused by a rule as a measure for its usefulness in an intensional definition. An experiment shows that this system performs quite well on a small example database.

Although the system can be used in a fully automatic way, it is more appropriate to think of it as an intelligent tool helping the database designer. Many

decisions made by the system are based on heuristics, and the system has a high chance of making the right decisions, but some supervision by the user is certainly recommended.

Our current system can still be extended in many directions. Topics for future research include:

- noise handling: at this moment, no exceptions to rules are allowed
- a comparison between several compaction criteria
- interactivity: at this moment the only way the user can influence the outcome of the system is through the language bias. There should be more opportunities to interfere with the design process.
- integrity constraints: we believe that the described techniques can also be used to find semantic integrity constraints for the database

Finally, we want to mention the work of Sommer [7], which is related to ours in the sense that both consider the problem of restructuring a knowledge base. The main difference is that in [7], existing rules are restructured, while our system can start from a purely extensional database.

Acknowledgements

This research is financed with a grant from the Flemish Institute for the Advancement of Scientific-Technological Research in the Industry (IWT), and it is also part of the European Community ESPRIT project no. 20237 (ILP2). Luc De Raedt is supported by the Belgian National Fund for Scientific Research.

References

1. L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.
2. L. De Raedt and L. Dehaspe. Clausal discovery. Submitted.
3. L. De Raedt and S. Džeroski. First order *jk*-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.
4. L. Dehaspe and L. De Raedt. DLAB: a declarative language bias formalism. This volume.
5. S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
6. G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
7. E. Sommer. Rulebase stratification: an approach to theory restructuring. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 377–390, Sankt Augustin, Germany, 1994. Gesellschaft für Mathematik und Datenverarbeitung MBH.
8. W. Van Laer, L. Dehaspe, and L. De Raedt. Applications of a logical discovery engine. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 263–274, 1994.