A Brief Introduction to Deductive Databases

Michael Genesereth
Computer Science Department
Stanford University

1. Introduction

A *deductive database* is a finite collection of facts and rules. By applying the rules of a deductive database to the facts in the database, it is possible to infer additional facts, i.e. facts that are implicitly true but are not explicitly represented in the database.

This paper is a brief introduction to deductive databases. In the first section, we talk about traditional databases, i.e. sets of simple facts. After that, we introduce logic programs, i.e. sets of rules. We then show how to use rules in defining views of a database, in writing constraints on the database, and in defining updates to the database. We close with a brief discussion of special, built-in functions and relations.

2. Sentential Databases

When we think about the world, we usually think in terms of objects and relationships among these objects. Objects include things like people and offices and buildings. Relationships include things like the parenthood, ancestry, office assignments, office locations, and so forth.

In sentential databases, we encode each instance of a relationship in the form of a *sentence* consisting of a *relation constant* representing the relationship and some *terms* representing the objects involved in the instance. For example, we could use the relation constant `parent` to represent the relationship between a parent and his or her child; we could use the terms `art` and `bob` to refer to two people; and using this vocabulary we could write a sentence stating that art is the parent of bob.

The *vocabulary* of a database is a collection of object constants, function constants, and relation constants. Each function constant and relation constant has an associated arity, i.e. the number of objects involved in any instance of the corresponding function or relation.

A *term* is either a symbol or a functional term. A *functional term* is an expression consisting of an *n*-ary function constant and *n* terms. In what follows, we write functional terms in traditional mathematical notation - the function followed by its *arguments* enclosed in parentheses and separated by commas. For example, if `f` is a binary function constant and if `a` and `b` are object constants, then `f(a,a)` and `f(a,b)` and `f(b,a)` and `f(b,b)` are all functional terms. Functional terms can be nested within other functional terms. For example, if `f(a,b)` is a functional term, then so is `f(f(a,b),b)`.

A *datum* is an expression formed from an *n*-ary relation constant and *n* terms. We write data in mathematical notation. For example, we could write `parent(art,bob)` to express the fact that Art is the parent of Bob.

A *database instance*, or *dataset*, is any set of data that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a database instance as the facts that we believe to be true in the world; data that are not in the instance are assumed to be false.

As an example of these concepts, consider a small interpersonal database. The terms in this case represent people. The relation constants name properties of these people and their relationships with each other.

In our example, we use the binary relation constant `parent` to specify that one person is a parent of another. The sentences below constitute a database describing six instances of the `parent` relation. The person named `art` is a parent of the person named `bob`; `art` is also a parent of `bea`, and so forth.

```
parent(art,bob)
parent(art,bea)
parent(bob,carl)
parent(bea,coe)
parent(carl,daisy)
parent(carl,daniel)
```

The `adult` relation is unary relation, i.e. a simple property of a person, not a relationship other people. Everyone in our database is an adult except for `daisy` and `daniel`.

```
adult(art)
adult(bob)
adult(bea)
adult(carl)
adult(coe)
```

We can express gender with two unary relation constants `male` and `female`. The following sentences expresses the genders of all of the people in our database. Note that, in principle, we need only one relation here, since one gender is the complement of the other. However, representing both allows us to enumerate instances of both genders equally efficiently, which can be useful in certain applications.

```
male(art)          female(bea)
male(bob)          female(coe)
male(cal)          female(daisy)
male(daniel)
```

As an example of a ternary relation, consider the data shown below. Here, we use `prefers` to represent the fact that the first person likes the second person more than the third person. For example, the first sentence says that Art prefers `bea` to `bob`; the second sentence says that `carl` prefers `daisy` to `daniel`.

```
prefers(art,bea,bob)
prefers(carl,daisy,daniel)
```

Note that the order of arguments in such sentences is important. Given the meaning of the `prefers` relation in our example, the first argument denotes the subject, the second argument is the person who is preferred, and the third argument denotes the person who is less preferred. Of course, we could equally well have interpreted the arguments in other orders. The important thing is consistency - once we choose to interpret the arguments in one way, we must stick to that interpretation everywhere.

3. Logic Programs

The rules in a deductive database are often called a *logic program*. The language of logic programs includes the language of databases but provides additional expressive features.

One key difference is the inclusion of a new type of symbol, called a *variable*. Variables allow us to state relationships among objects without explicitly naming those objects. In what follows, we use individual capital letters as variables, e.g. `X`, `Y`, `Z`.

In the context of logic programs, a *term* is defined as an object constant, a variable, or a functional term, i.e. an expression consisting of an *n*-ary function constant and *n* simpler terms.

An *atom* in a logic program is analogous to a datum in a database except that the constituent terms may include variables.

A *literal* is either an atom or a negation of an atom (i.e. an expression stating that the atom is false). A simple atom is called a *positive* literal, The negation of an atom is called a *negative* literal. In what follows, we write negative literals using the negation sign ~. For example, if `p(a,b)` is an atom, then `~p(a,b)` denotes the negation of this atom.

A *rule* is an expression consisting of a distinguished atom, called the *head* and a conjunction of zero or more literals, called the *body*. The literals in the body are called *subgoals*. In what follows, we write rules as in the example shown below. Here, `r(X,Y)` is the head, `p(X,Y) & ~q(Y)` is the body; and `p(X,Y)` and `~q(Y)` are subgoals.

```
r(X,Y) :- p(X,Y) & ~q(Y)
```

Semantically, a rule is something like a reverse implication. It is a statement that the conclusion of the rule is true whenever the conditions are true. For example, the rule above states that `r` is true of any object `X` and any object `Y` *if* `p` is true of `X` and `Y` and `q` is not true of `Y`. For example, if we know `p(a,b)` and we know that `q(b)` is false, then, using this rule, we can conclude that `r(a,b)` must be true.

Exercise: Click here to test your understanding of rule syntax.

A *logic program* is a finite set of atoms and rules as just defined. In order to simplify our definitions and analysis, we occasionally talk about infinite sets of rules. While these sets are useful, they are not themselves logic programs.

Unfortunately, the language of rules, as defined so far, allows for logic programs with some unpleasant properties. To avoid programs of this sort, it is common in deductive databases to add a couple of restrictions that together eliminate these problems.

The first restriction is *safety*. A rule in a logic program is *safe* if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body. A logic program is safe if and only if every rule in the program is safe.

The example shown above is safe. By contrast, the two rules shown below are not safe. The first rule is not safe because the variable `Z` appears in the head but does not appear in any positive subgoal. The second rule is not safe because the variable `Z` appears in a negative subgoal but not in any positive subgoal.

```
s(X,Y,Z) :- p(X,Y)
t(X,Y) :- p(X,Y) & ~q(Y,Z)
```

To see why safety is matters in the case of the first rule, suppose we had a database in which `p(a,b)` is true. Then, the body of the first rule is satisfied if we let `X` be `a` and `Y` be `b`. In this case, we can conclude that every corresponding instance of the head is true. But what should we substitute for `Z`? Intuitively, we could put anything there; but there could be infinitely many possibilities. While this is conceptually okay, it is practically problematic.

To see why safety matters in the second rule, suppose we had a database with just two facts, viz. `p(a,b)` and `q(b,c)`. In this case, if we let `X` be `a` and `Y` be `b` and `Z` be anything other than `c`, then both subgoals true, and we can conclude `t(a,b)`. The main problem with this is that many people incorrectly interpret that negation as meaning there is no `Z` for which `q(Y,Z)` is true, whereas the correct reading is that `q(Y,Z)` needs to be false for just one binding of `Z`. As we will see in our examples below, there is a simple way of expressing this other meaning without writing unsafe rules.

Exercise: Click here to test your understanding of the concept of safety.

The second restriction is called *stratified negation*. It is essential in order to avoid ambiguities. Unfortunately, it is a little more difficult to understand than safety.

The *dependency graph* for a logic program is a directed graph with two type of arcs, *positive* and *negative*. The nodes in the dependency graph for a program represent the relations in the program. There is a positive arc in the graph from one node to another if and only if the former node appears in a positive subgoal of a rule in which the latter node appears in the head. There is a negative arc from one node to another if and only if the former node appears in a negative subgoal of a rule in which the latter node appears in the head.

As an example, consider the following logic program. `r(X,Y)` is true if `p(X,Y)` and `q(Y)` are true. `s(X,Y)` is true if `r(X,Y)` is true and `s(Y,X)` is false.

```
r(X,Y) :- p(X,Y) & q(Y)
s(X,Y) :- r(X,Y) & ~s(Y,X)
```

The dependency graph for this program contains nodes for `p`, `q`, `r`, and `s`. Due to the first rule, there is a positive arc from `p` to `r` and a positive arc from `q` to `r`. Due to the second rule, there is a positive arc from `r` to `s` and a negative arc from `s` to itself.

A negation in a logic program is said to be *stratified with respect to negation* if and only if there is no negative arc in any cycle in the dependency graph. The logic program just shown is *not* stratified with respect to negation because there is a cycle involving a negative arc.

The problem with unstratified logic programs is that there is a potential ambiguity. As an example, consider the program above and assume we had a database containing `p(a,b)`, `p(b,a)`, `q(a)`, and `q(b)`. From these facts we can conclude `r(a,b)` and `r(b,a)` are both true. So far so good. But what can we say about `s`? If we take `s(a,b)` to be true and `s(b,a)` to be false, then the second rule is satisfied. If we take `s(a,b)` to be false and `s(b,a)` to be true, then the second rule is again satisfied. We can also take them both to be true. The upshot is that there is ambiguity about `s`. By concentrating exclusively on programs that are stratified with respect to negation, we avoid such ambiguities.

Exercise: Click here to test your understanding of the concept of stratified negation.

In writing logic programs, we avoid these problems by ensuring that all our programs are both safe and stratified with respect to negation. The restrictions are easy to satisfy in most applications; and, by obeying these restrictions, we ensure that our logic programs produce well-defined answers.

4. Views and Queries

The principal use of rules is to define new relations in terms of existing relations. The new relations defined in this way are often called *view relations* (or simply views) to distinguish them from *base relations*, which are defined by explicit enumeration of instances.

To illustrate the use of rules in defining views, consider once again the world of interpersonal relations. Starting with the base relations, we can define various interesting view relations.

As an example, consider the sentences shown below. The first sentence defines the `father` relation in terms of `parent` and `male`. The second sentence defines `mother` in terms of `parent` and `female`.

```
father(X,Y) :- parent(X,Y) & male(X)
mother(X,Y) :- parent(X,Y) & female(X)
```

The rule below defines the grandparent relation in terms of the parent relation. A person `x` is the grandparent of a person `z` if `x` is the parent of a person `y` and `y` is the parent of `z`. The variable `y` here is a *thread variable* that connects the first subgoal to the second but does not itself appear in the head of the rule.

```
grandparent(X,Z) :- parent(X,Y) & parent(Y,Z)
```

Note that the same relation can appear in the head of more than one rule. For example, the `person` relation is true of a person `y` if there is an `x` such that `x` is the parent of `y` *or* if `y` is the parent of some person `z`. Note that in this case the conditions are disjunctive (at least one must be true), whereas the conditions in the grandfather case are conjunctive (both must be true).

```
person(X) :- parent(X,Y)
person(Y) :- parent(X,Y)
```

A person `x` is an ancestor of a person `z` if `x` is the parent of `z` or if there is a person `y` such that `x` is an ancestor of and `y` is an ancestor of `z`. This example shows that is possible for a relation to appear in its own definition. (But recall our discussion of stratification for a restriction on this capability.)

```
ancestor(X,Y) :- parent(X,Y)
ancestor(X,Z) :- ancestor(X,Y) & ancestor(Y,Z)
```

A childless person is one who has no children. We can define the property of being childless with the rules shown below. The first rule states that a person `x` is childless if `x` is a person and it is not the case that `x` is a parent. The second rule says that `isparent` is true of `x` if `x` is the parent of some person `y`.

```
childless(X) :- person(X) & ~isparent(X,Y)
isparent(X) :- parent(X,Y)
```

Note the use of the helper relation `isparent` here. It is tempting to write the childless rule as `childless(X) :- person(X) & ~parent(X,Y)`. However, this would be wrong. This would define `x` to be childless if `x` is a person and there is *some* `y` such that `x` is `~parent(X,Y)` is true. But we really want to say that `~parent(X,Y)` holds for *all* `y`. Defining `isparent` and using its negation in the definition of `childless` allows us to express this *universal quantification*.

Exercise: Click here to familiarize yourself with rules by defining these relations and others in an interactive editor.

In practice, view definitions are usually stored in a deductive database for later use. However, another common use is to define one-shot queries on the database. Such

As a simple example, suppose we wanted to know all people who are grandparents of adults. We could ask this question by writing the query shown below.

```
query(Z) :- grandparent(art,Z) & adult(Z)
```

By including definitions for auxiliary temporary relations, we can also ask more complicate queries. For example, we might want to know all people who are parents of childless adults. We can ask this question by (1) defining a temporary relation isparent that is true of any person who has a parent and (2) adding this condition as a negative subgoal in the query.

```
query(X) :- parent(X,Y) & adult(Y) & ~isparent(Y)
isparent(Y) :- parent(Y,Z)
```

What makes these queries different from ordinary view definitions is that they are temporary - they last for the duration of a query and are then discarded. This avoids filling up the database with

redundant information.

5. Static Constraints

In our development thus far, we have assumed that the extension of an *n*-ary relation may be any set of *n*-tuples from the domain. This is rarely the case. Often, there are *constraints* that limit the set of possibilities. For example, a person cannot be his own parent. In some cases, constraints involve multiple relations. For example, all parents are adults; in other words, if an entity appears in the first column of the `parent` relation, it must also appear as an entry in the `adult` relation.

In many database texts, constraints are written in direct form - by writing rules that say, in effect, that if certain things are true in an extension, then other things must also be true. The *inclusion dependency* mentioned above is an example - if an entity appears in the first column of the `parent` relation, it must also appear as an entry in the `adult` relation.

In what follows, we use a slightly less direct approach - we encode limitations by writing rules that say when a database is *not* well-formed. We simply invent a new 0-ary relation, here called `illegal`, and define it to be true in any extension that does not satisfy our constraints.

This approach works particularly well for consistency constraints like the one stating that a person cannot be his own parent.

```
illegal :- parent(X,X)
```

It also works well for *mutual exclusion* constraints like the one below, which states that a person cannot be in both the `male` and the `female` relations.

```
illegal :- male(X) & female(X)
```

Using this technique, we can also write the *inclusion dependency* mentioned earlier. There is an error if an entity is in the first column of the `parent` relation and it does not occur in the `adult` relation.

```
illegal :- parent(X,Y) & ~adult(X)
```

Database management systems can use such constraints in a variety of ways. They can be used to optimize the processing of queries. They can also be used to check that updates do not lead to unacceptable extensions.

6. Updates

In updating a database, a user specifies a sentence to add to a database or a sentences to delete. In some cases, the user can group several changes of this sort in a single, so-called, *atomic* transaction. If the result of executing the transaction satisfies the constraints, the update is performed; otherwise it is rejected.

Unfortunately, if a user forgets to include an addition or deletion required by the constraints, this can lead to errors. In order to simplify the update process for the user, some database systems provide the administrator the ability to write update rules, i.e. rules that are executed by the system to augment a specified transaction with the additions and deletions necessary to avoid errors. In what follows, we show one way that this can be done

Our update language includes four special operators - pluss, minus, pos, and neg. pluss takes a sentence as argument and is true if and only if the *user* specifies that sentence as an addition in a transaction. minus takes a sentence as argument and is true if and only if the *user* specifies that sentence as an addition in a transaction. pos takes a sentence as argument and is true if and only if the *system* concludes that the specified sentence should be added to the database. neg takes a sentence as argument and is true if and only if the *system* concludes that the specified sentence

should be added to the database. Update rules are rules that define pos and neg in terms of pluss and minus and the current state of the database.

As an example of this mechanism in action, consider the rules shown below. The first dictates that the system remove a sentence of the form `male(X)` whenever the user adds a sentence of the form `female(X)`. The second rule is analogous to the first with `male` and `female` reversed. Together, these two rules enforce the mutual exclusion on `male` and `female`.

```
neg(male(X)) :- pluss(female(X))
neg(female(X)) :- pluss(male(X))
```

Similarly, we can enforce the inclusion dependency on `parent` and `adult` by writing the following rule. If the user adds a sentence of the form `parent(X,Y)`, then the system also adds a sentence of the form `adult(X)`.

```
pos(adult(X)) :- pluss(parent(X,Y))
```

Another use of this update mechanism is to maintain materialized views. (A *materialized view* is a defined relation that is stored explicitly in the database, usually to save recomputation.)

Suppose, for example, we were to materialize the `father` relation defined earlier. Then we could write the update rules to maintain this materialized view. According to the first rule, the system should add a sentence of the form `father(X,Y)` whenever the user adds `parent(X,Y)` and `male(X)` is know to be true and the user does not delete that fact. The other rules cover the other cases.

```
pos(father(X,Y)) :- pluss(parent(X,Y)) & male(X) & ~minus(male(X))
pos(father(X,Y)) :- parent(X,Y) & pluss(male(X)) & ~minus(parent(X,Y))
pos(father(X,Y)) :- pluss(parent(X,Y)) & pluss(male(X))
neg(father(X,Y)) :- minus(parent(X,Y))
neg(father(X,Y)) :- minus(male(X))
```

Note that not all constraints can be enforced using update rules. For example, if a user suggests adding the sentence `parent(art,art)` to the database in our interpersonal relations example, there is nothing the system can do to repair this error except to reject the transaction. In some cases, there is no way to make a repair unambiguously; more information is needed from the user. For example, we might have a constraint that every person is in either the male or the female relation. If the user specifies a parent fact involving a new person but does not specify the gender of that person, there is no way for the system to decide that gender for itself.

7. Special Relations

In practical logic programming languages, it is common to "build in" commonly used concepts. These typically include arithmetic functions (such as `+`, `*`, `max`, `min`), string functions (such as concatenation), comparison operators (such as < and >), and equality (=). It is also common to include aggregate operators, such as `countofall`, `avgofall` `sumofall`, and so forth.

In many practical logic programming languages, mathematical functions are represented as relations. For example, the the binary addition operator + is often represented by the the ternary relation constant `plus`. For example, the following rule defines the combined age of two people. The combined age of `X` and `Y` is `S` if the age of `X` is `M` and the age of `Y` is `N` and `S` is the result of adding `M` and`N`.

```
combinedage(X,Y,S) :- age(X,M) & age(Y,N) & plus(M,N,S)
```

Similarly, aggregate operators are typically represented as relations. For example the following rule defines the number of a person's grandchildren using the `countofall` relation in this way. `N` is

the number of grandchildren of `X` if `N` is the count of all `Z` such that `X` is the grandparent of `Z`.

```
grandchildren(X,N) :- person(X) & countofall(Z,grandparent(X,Z),N)
```

In logic programming languages that provide such built-in concepts, there are usually syntactic restrictions on their use. For example, if a rule contains a subgoal with a comparison relation, then every variable that occurs in that subgoal must occur in at least one positive literal in the body and that occurrence must precede the subgoal with the comparison relation. If a rule mentions an arithmetic function, then any variable that occurs in all but the last position of that subgoal must occur in at least one positive literal in the body and that occurrence must precede the subgoal with the arithmetic relation.