DEDUCTIVE DATABASES

April 26th, 2002

Michael Minock

Deductive Databases

At the intersection of Logic, Databases, and Artificial Intelligence

Specify *rules* in a declarative language

Inference engine deduces new *facts* from old facts

Data model is essentially relational (Domain Relational Calculus)

The declarative knowledge model is based on the Prolog programming language

Together, this relational **data** model and PRO**LOG** like knowledge model form:**Datalog**.

Specification of a Deductive Database

- The facts:
  - written as tuples, but we leave off attribute names
  - meaning of a tuple value based on its position
- The rules:
  - similar to (ideal) relational views - specify a virtual relations
  - rules may be applied recursively – not possible with views
  - special (practical) features in systems

Overview

- Datalog Notation
- Interpretation of Datalog Programs
- Inference Mechanisms (Bottom-up vs. Top-down)
- Datalog Programs and Their Evaluation
- Deductive Database Systems

Datalog Predicate Notation

A *predicate* with a fixed number of *arguments*

If the arguments are all constant values then the predicate simply states that a fact is true.

By convention constant values are in lowercase or numeric form.

We say that such a predicate is a *g*round predicate.

variable names, in capital letters, reference attribute in the corresponding position, and are able to range over a set (possibly infinite) of constant values.

A predicate with one or more variables in it, is said to be a *non-ground* predicate.

Figure 25.1 (a) Prolog notation. (b) The supervisory tree.

(a)

Facts

```
supervise(franklin,john).
supervise(franklin,ramesh).
supervise(franklin,joyce).
supervise(jennifer,alicia).
supervise(jennifer,ahmad).
supervise(james,franklin).
supervise(james,jennifer).
    ...
```

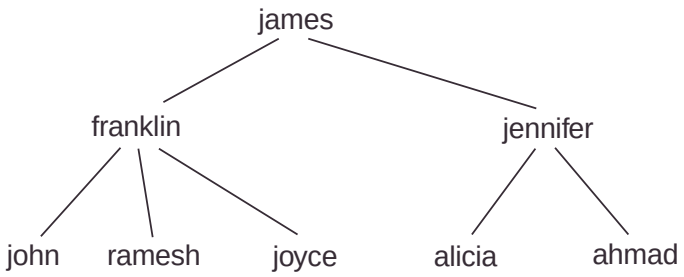
Rules

```
superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).
subordinate(X,Y) :- superior(Y,X).
```

Queries

```
superior(james,Y)?
superior(james,joyce)?
```

(b)



Example

The predicate `supervise(X,Y)` states that "X supervises Y"

A rule is specified in the form *head :- body* where ':-' is read 'if' !

The head is a single predicate

The body consists of a list of predicates.

If a collection of *bindings* (or assignments) of constant values to the variables of the body of the rule make all the body predicates true, this causes, using this same collection of bindings, the head of the rule to be true. Thus inducing new facts.

Datalog Rule Notation

Read the ','s in the body as logical 'and'.

Read the same head predicate defined in two separate rules as logical 'or'.

Notice that the rules may be recursive. Note that this is what happens with `superior(X,Y)` in our example.

We may use built-in predicates such as $=, <, >, \ldots$ in the body.

A *query* is a predicate. Variables in the query is a request for all of the value combinations that make the predicate true.

Formal Notation

Now we will adopt a bit more of a formal outlook.

The basic building blocks are predicates of the form: $p(a_1, ..., a_n)$, where p is said to be of the degree or arity n.

A *literal* is a predicate with a possible negation $\neg$ symbol modifying it.

Datalog programs may represent a proper subset of FOPC.

Clausal Form

We may put any expression in FOPC into clausal form. (Name the algorithm...)

All variables must be universally quantified Therefor the $\forall$ symbol is implicit.

A clause is simply a disjuction of literals.

A **conjunctive normal form** (CNF) formula is made up of a number of conjoined clauses.

$(l_{1,1} \vee ... \vee l_{1,n_1}) \wedge ... \wedge (l_{m,1} \vee ... \vee l_{m,n_m})$

This is what your rule bases are. Just one big CNF!

With two primary restrictions: (basically) function free plus clauses are Horn!

Horn Clauses

Consider the clause:

$\neg p_1 \vee ... \vee \neg p_n \vee q_1 \vee ... \vee q_m$

This may be converted into:

$p_1 \wedge ... \wedge p_n \Rightarrow q_1 \vee ... \vee q_n$

Horn clauses are clauses in which there is at most 1 positive literal.

if $m = 1$ this gives us $p_1 \wedge ... \wedge p_n \Rightarrow q_1$

if $m = 0$ this gives us $p_1 \wedge ... \wedge p_n \Rightarrow$

The later construct illustrates an integrity constraint.

Don't get me started on what q-Horn form is, and what the Horn-core is and what a Horn envelope of a CNF is.

A Query Run Over Datalog

A set of ground predicates (facts).

A set of function-free conjoined horn clauses (rules).

A literal $p(X_1, ..., X_n)$ (query).

The system has an internal **inference engine** that computes the set of all *c*orrect constant bindings to the variables $X_1, ..., X_n$. In Datalog, this whole set is returned. In Prolog only one binding is returned at a time.

The default reasoning folks call this *a*nswer set semantics.

Interpretation of Datalog Programs

Two main ways of determining the semantics of programs:

proof-theoretic and *model-theoretic*

In the proof-theoretic interpretation, the facts and rules are true statements (**axioms**)

The facts are ground axioms that are given to be true.

Rules are **deductive axioms** that enable us to deduce additional facts to be true.

The answer to the query of whether a fact follows from the given facts may be achieved through deduction.

The answer to the query of whether a fact does not follow from the given facts requires deduction and an assumption - more on this later.

In the proof theoretic interpretation rules are 'applied' syntactically. A well defined semantics lets us do this without worry. (Think of the analogy with Calculus and the act of differentiation)

Model Theoretic Interpretation

In the Model-theoretic interpretation, we have a finite (in the *Herbrand case*) domain of constants for each attribute.

We consider every combination of values as arguments.

We then consider, for every combination, whether the predicate is true or false.

This is how we specify a *w*orld ω (or an *i*nterpretation).

A subset of the possible worlds (or interpretations) may be models of the program.

The world ω is a **model** for a set of facts and rules if no facts or rules are falsified by ω.

$\phi \rightarrow \varphi$ is falsified by ω if $\omega \models \phi \wedge \neg\varphi$.

Figure 25.3 An interpretation that is a minimal model.

Rules

superior(X,Y) :- supervise(X,Y).
superior(X,Y) :- supervise(X,Z), superior(Z,Y).

Interpretation

Known Facts:

supervise(franklin,john) is **true**.
supervise(franklin,ramesh) is **true**.
supervise(franklin,joyce) is **true**.
supervise(jennifer,alicia) is **true**.
supervise(jennifer,ahmad) is **true**.
supervise(james,franklin) is **true**.
supervise(james,jennifer) is **true**.
supervise(X,Y) is **false** for all other possible (X,Y) combinations.

Derived Facts:

superior(franklin,john) is **true**.
superior(franklin,ramesh) is **true**.
superior(franklin,joyce) is **true**.
superior(jennifer,alicia) is **true**.
superior(jennifer,ahmad) is **true**.
superior(james,franklin) is **true**.
superior(james,jennifer) is **true**.
superior(james,john) is **true**.
superior(james,ramesh) is **true**.
superior(james,joyce) is **true**.
superior(james,alicia) is **true**.
superior(james,ahmad) is **true**.
superior(X,Y) is **false** for all other possible (X,Y) combinations.

Minimal Model

We are usually most interested in the **minimal model**.

Suppose if we add the fact `superior(james,bob)`

We still have a model, but not (a) the minimal one

For the simple (non-negation) case. The proof-theoretic and model theoretic interpretations coincide. But, once you admit negation, the two views diverge.

The Computational Interpretation

We must ultimately build a decision mechanism to answer queries.

We would like this mechanism to coincide with a well-defined semantic interpretation.

Normally a proof theoretic interpretation guides this development.

But such an proof system need a control strategy.

The basic approaches are either: Bottom-up or top-down.

Bottom-up Inferencing (Forward Chaining)

the engine takes a set of facts and recursively applies the rules until the set of facts grow no more.

But we must be more clever than this. We are only interested in the query we asked. Not the entire set of facts - potentially infinite - that could be deduced.

The basic proof rule is *modus ponens*: $p \rightarrow q$, p therefore q.

Classical AI forward Chainers (based on the RETE - many pattern/ many object match) use *conflict resolution* - OPS5 and CLIPS

Top-down Inferencing (Backward Chaining)

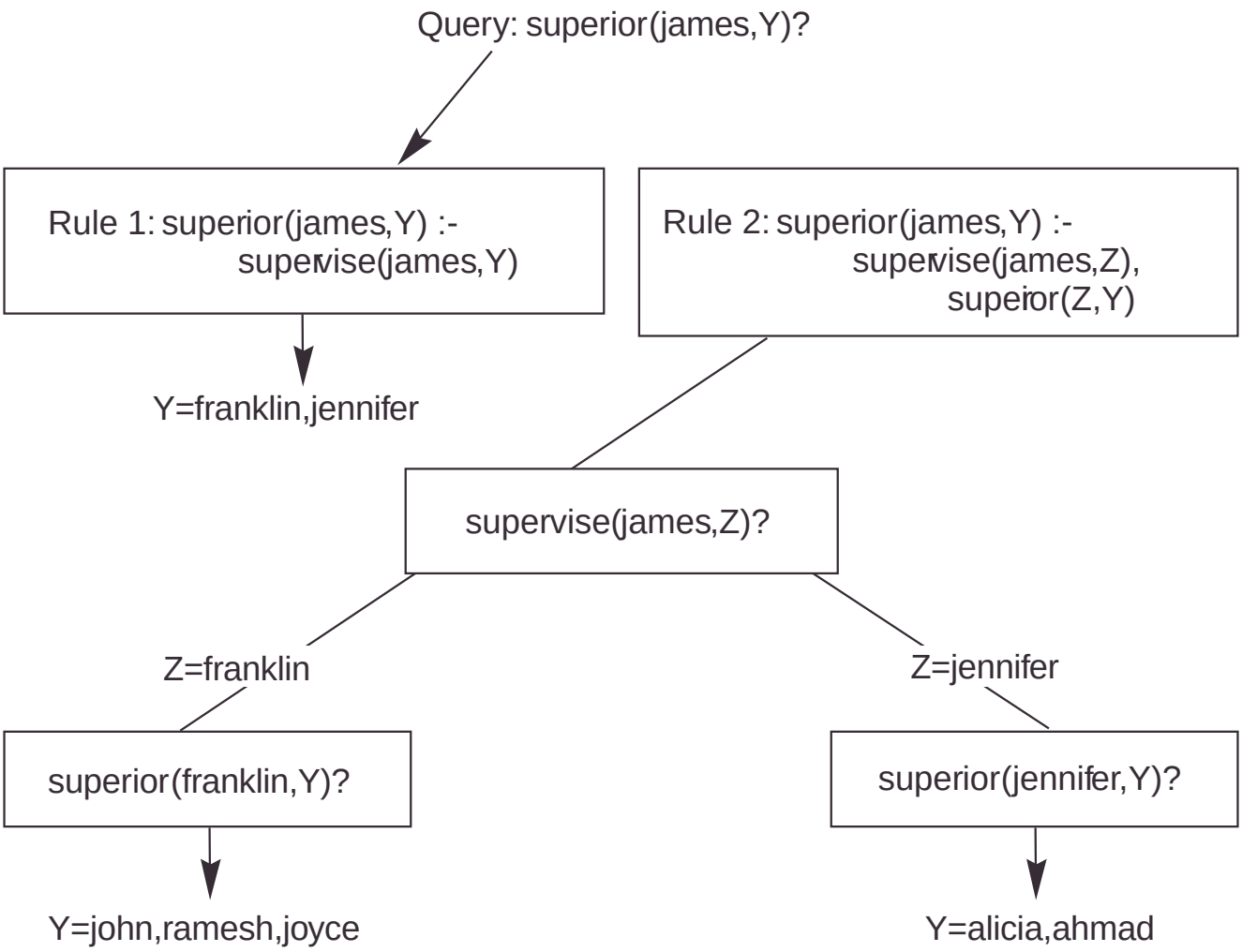
Starting with the query predicate, attempt to find matches to variables that lead to valid facts in the database.

Consider facts and rules in some (arbitrary) order

Maintain coherent variable binding structure and grow the tree downward, branching **sub-goals**, looking to eventually satisfy predicates with ground facts.

Conduct the search either depth-first or breath-first.

Figure 25.4 Top-down evaluation of a query.



Top-down Inferencing (Backward Chaining)

Sensitive to the arbitrary order of rules - possible infinite recursion

```
a(x,z) :- a(x,y), p(y,z).
a(x,y) :- p(x,y).
```

Negation is also difficult to handle

Datalog Programs and Their Evaluation

Two types of predicates: Fact-based (or relations) or Rule-defined predicates (or views).

The fact based predicates are said to belong to the **EDB** (Extensional Database)

The rule defined predicates are said to belong to the **IDB** (Intensional Database)

These term *extensional* and *intensional* come from the philosophy of language.

Safety

Rules must be safe. That is they must generate only a finite set of facts

```
big_salary(Y) :- Y>60000
big_salary(Y) :- employee(X),salary(X,Y),Y>60000
big_salary(Y) :- Y>60000, employee(X),salary(X,Y)
```

A variable is limited if either

- (1) it appears in a regular, not built-in predicate of a rule body.
- (2) it appears in a predicate of the form X=c or c=X, where c is a constant , in the rule body.
- (3) it appears is a predicate of the form X=Y where Y is a limited variable, in the rule body.

A rule is safe, if all of its variables are limited.

Relational Operators

Figure 25.7 Predicates for illustrating relational operations.

```
rel_one(A,B,C).
rel_two(D,E,F).
rel_three(G,H,I,J).

select_one_A_eq_c(X,Y,Z) :- rel_one(c,Y,Z).
select_one_B_less_5(X,Y,Z) :- rel_one(X,Y,Z), Y<5.
select_one_A_eq_c_and_B_less_5(X,Y,Z) :- rel_one(c,Y,Z), Y<5.

select_one_A_eq_c_or_B_less_5(X,Y,Z) :- rel_one(c,Y,Z).
select_one_A_eq_c_or_B_less_5(X,Y,Z) :- rel_one(X,Y,Z), Y<5.

project_three_on_G_H(W,X) :- rel_three(W,X,Y,Z).

union_one_two(X,Y,Z) :- rel_one(X,Y,Z).
union_one_two(X,Y,Z) :- rel_two(X,Y,Z).

intersect_one_two(X,Y,Z) :- rel_one(X,Y,Z), rel_two(X,Y,Z).

difference_two_one(X,Y,Z) :- rel_two(X,Y,Z), not(rel_one(X,Y,Z)).

cart_prod_one_three(T,U,V,W,X,Y,Z) :-
                              rel_one(T,U,V), rel_three(W,X,Y,Z).

natural_join_one_three_C_eq_G(U,V,W,X,Y,Z) :-
                               rel_one(U,V,W), rel_three(W,X,Y,Z).
```

Evaluation of Non-Recursive Datalog Queries

Bottom-up use of the relational operators in Datalog.

If the query involves only fact-based predicates, then we may solve the query through a simple relational algebra query expression.

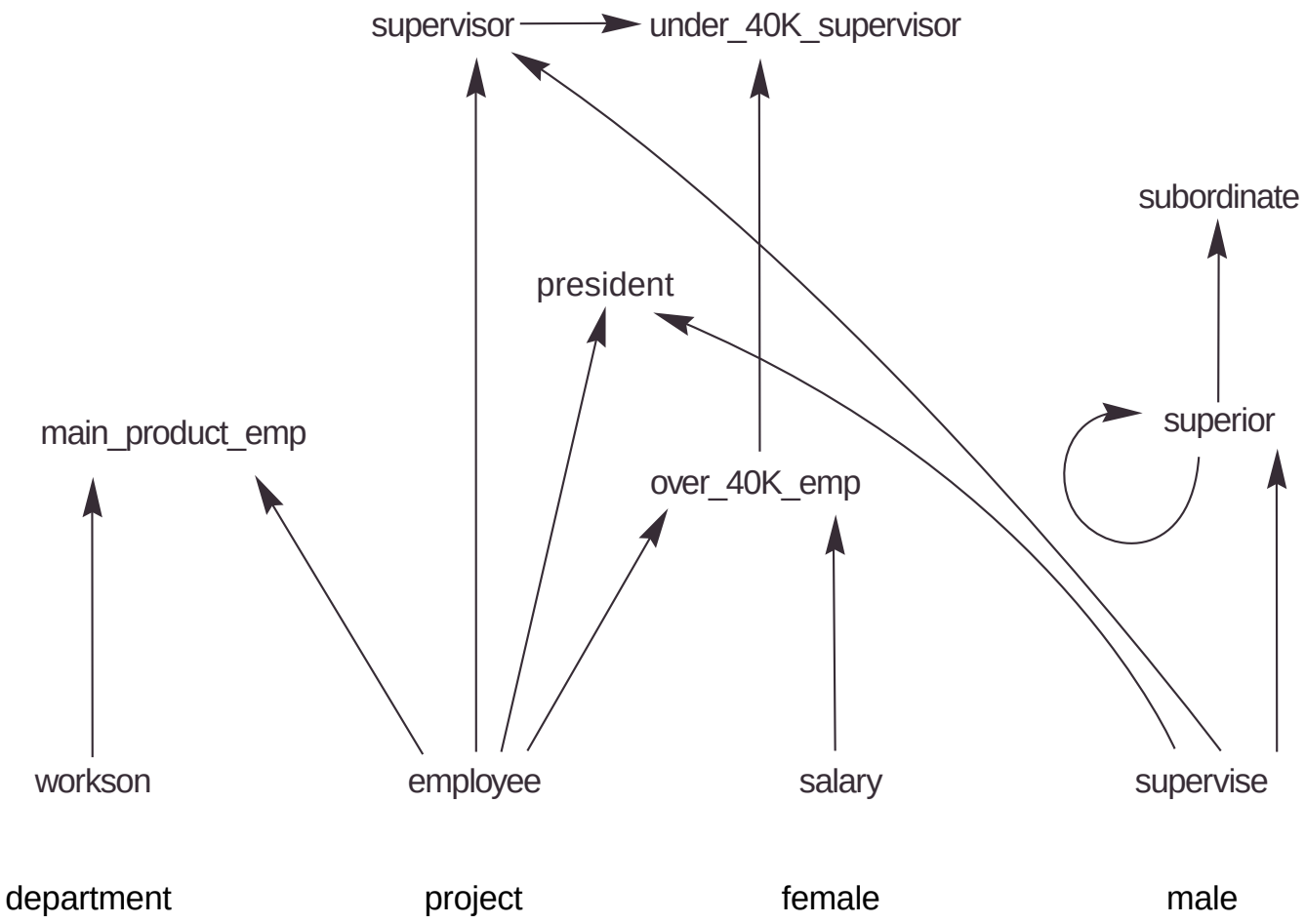
When a query involves a rule-defined predicate

$p :- p_1, ..., p_n$ the strategy is to first compute the results for the $p_1, ..., p_n$ relations. And then for p.

A predicate dependency graph is of some use to determine the nodes where answers will need to be materialized.

Predicate Dependency Graph

Figure 25.8 Predicate dependency graph for Figures 25.6 and 25.7.



Non-Recursive Queries

A non-recursive query is one that includes only non-recursive predicates

Based on the head predicate of our query we compute a partial ordering of the predicates upon which our goal predicate depends.

Note that this process terminates with a set of fact-based predicates.

Step 1: generate a 'query plan' top-down

Step 2: evaluate the query plan, bottom-up

Rectified Rules

We also convert our rules into **rectified rules**. A rule is rectified if all the variables in the head of the rule are distinct.

Introduce new predicates in the body of the rule, to handle repeating variables and constants.

```
knows(X,X,'well') :- Person(X), Thoughtful(X).
```

becomes:

```
knows(X,Y,Z) :- Person(X), Thoughtful(Y),
  X=Y, Z = 'well'.
```

Evaluation of Non-Recursive Predicates

$q = p(arg_1, ..., arg_n)$

If p is fact-based: Generate the expression

```
Select *
From P
Where
  <conditions>
```

Where conditions contains:

$att_i = C$ if arg_i is the constant value C.

$att_i = att_j$ if arg_i and arg_j are the same variable

Evaluation of Non-Recursive Predicates

If p is predicate-based:

Collect all of the rules $S_i, i = 1, ..., n$ which has predicate p as their head.

for each rule S_i:
Re-apply this entire algorithm to each predicate in the rule body - with constant variables substituted and equal variable resolved to the same name.

For book keeping it is perhaps useful to create a temporary table to contain the results.

Perform a natural join amount the predicates of the rule S_i. This yields the relation R_s

Apply built in predicates of the rule to the result and project out the variables of the rule head.

Evaluation of Non-Recursive Predicates

The resulting expressions from each rule are are all unioned together.

Note that this algorithm produces a bottom-up evaluation plan with constants properly substituted.

Notice that the algorithm is non-optimal.

Recursive Queries

Pure evaluation approach: Naive and Semi-Naive approaches

Rule rewriting approach: Optimizing the plan into a more efficient strategy: Magic-sets

Linear Recursion

```
(1) ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
(2) ancestor(X,Y) :- parent(X,Y).
```

(1) is **left linear recursive**

```
(1') ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

(1') is **right linear recursive**

```
(1'') ancestor(X,Y) :- ancestor(X,Z),ancestor(Z,Y).
```

(1") is not linearly recursive.

As the book says, however, most real-life rules can be made linearly recursive. Can you think of some practical examples where this does not hold?

Fixed Points

We may replace ':-' with '=' and borrow from recursive function theory.

As we apply these functions we are present with a solution to the equations through time.

For simple datalog (without negation) the set of solution grows *monotonically*.

For safe rules constructed over finite domains, we will reach a fixed point, where the solution set does not grow.

This is the least fix-point solution to the application of the recursive functions.

Notation Preliminaries

The EDB relations: $q_1, ..., q_m$

The IDB relations: $r_1, ..., r_n$

The rule equations for the i-th rule are E_i

So a single application of the rules is:

$r_i = E_i(r_1, ..., r_n, q_1, ..., q_m)$

Jacobi

```
for  i=1 to n do Ri = null
 repeat
  condition = true;
  for i= 1 to n do Si = Ri
  for i= 1 to n do
    begin
      Ri = Ei(S1,...Sn, Q1,..., Qm)
      if Ri != Si condition = false
    end
until condition
```

Gauss-Seidel: immediate substitution of Ri into Si so that new values may be immediately used for Rj $j > i$.

Deltas

In the Jacobi method let:

$$D_i^k = R_i^k - R_i^{k-1}$$

This is the differential that is computed at the step i.

When the whole system of rules is linear, we may use just this *delta* at each step.

Semi-Naive

```
for i=1 to n do Ri=null
for i=1 to n do Di=null
 repeat
  for i=1 to n do Si = null
  condition = true
  for i = 1 to n do
    begin
      Di = Ei(S1,...Sn,Q1,...Qm) - Ri
      Ri = Di Union Ri
      if Di != null then condition = false
    end
 until condition
```

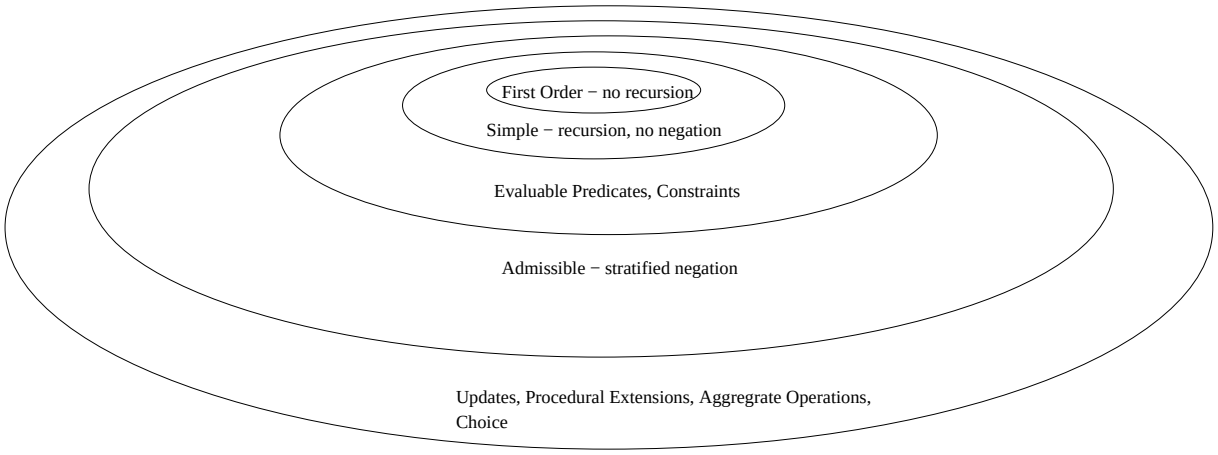
Magic Sets

```
sameGeneration(X,X).
sameGeneration(X,Y) :- parent(X,Z1),
                       parent(Y,Z2),
                       sameGeneration(Z1,Z2).
?sameGeneration(john,X).
```

combining the advantages of bottom-up and top-down, rewrite the rules:

```
sameGeneration(X,Y) :- magic(X),
                       parent(X,Z1),
                       parent(Y,Z2),
                       sameGeneration(X,Y).
sameGeneration(X,X) :- magic(X).
magic(john).
magic(U) :- magic(V),parent(V,U).
```

Hierarchy of Expressiveness



Relational Algebra $\equiv$ Tuple Calculus $\equiv$ Domain Calculus $\equiv$ First Order $+$ negation.

Negation and Logic

There is an old saying, "it is impossible to prove a negative."

Usually stated in the case of theological or legal questions.

Can negation make problems harder or impossible? And should we take this old saying literally? 'Yes' to the first and 'no' to the second.

Negation is hard because it often calls for us to range over an infinite set.

Assuming an infinite number of objects in the universe, the predicate $\neg InThisBox(X)$ will have infinite extent while the predicate $InThisBox(X)$ (perhaps) has finite extent.

Interpretation of Negation in Rules

```
single(X) :- !married(X)
```

Over the Herbrand universe {mike} we have two minimal models: {married(mike)} and {single(mike)} from the Herbrand Base: {{married(mike), single(mike)}, {married(mike)},{single(mike)},∅}

Given just the one deductive axiom above, should one minimal model be preferred over the other?

We would like a way of handling negations that:

1.) Makes sense to the user of the rules

2.) Allows for efficient answering of queries

Our answer (in this class), will be the concept of stratified negation.

Asymmetric Disjunction

Note that we shall not adequately handle the notion of representing the plain unbiased disjunction:

```
drinksSoda(X) or drinksBeer(X)
```

Under declarative semantics this is captured in the rule:

```
drinksSoda(X) :- !drinksBeer(X).
```

Under a constructive semantics we deduce that someone drinks soda if we find that they do not drink beer.

But we do not achieve the reverse deduction that someone drinks beer if they are found to not be drinking soda.

One solution would be to use two rules to express the disjunction. But this violates Stratified Negation.

Stratified Negation

"A program is stratified if there is no recursion through negation."

We induce the relations over predicates $\geq$ and $>$.

$p \geq q$ if we have the rule: $p \leftarrow ..., q, ...$

$p > q$ if we have the rule: $p \leftarrow ..., \neg q, ...$

A program P is admissible if there is no sequence of predicate symbols of P in the form

$p_1 \theta_1 p_2 ... \theta_{k-1} p_k$

such that $\theta \in \{\geq, >\}$, at least one $\theta_j =' >'$ and $p_1 = p_k$.

Stratified Negation

For a stratified program we may form a partitions of the predicates based on the $>$ and $\geq$ relations.

These partitions are called layers and there is a non-reflexive, anti-symmetric, transitive ordering $\succ$ over these layers.

Layers lower in the ordering $\succ$ propagate their answers to higher layers.

Before we propagate results to a higher layer i we calculate the fixed-points for all groups j where $i \succ j$.

Let us illustrate this with some examples.

Non-monotonicity

One final theoretical point to be made revolves around questions of monotonicity.

In classical logic the set of facts deduced grows monotonically as we learn new facts.

if $p \vdash q$ then $p \wedge r \vdash q$.

Note that we have monotonicity for first order and simple programs. Negation kills these monotonic properties.

The non-monotonic formalism underlying stratified negation is called *negation by failure*. It is in essence equivalent to the other 1st generation non-monotonic formalisms like simple *circumscription*(McCarthy, 1980) and *the closed world assumption*(Reiter, 1980).

Conclusion

This concludes our treatment of the theory of Deductive database systems.

What to remember:

Unlike prolog, we return a whole set at a time

Top-down query plan with bottom-up evaluation is the way to go.

Negation needs a structured treatment, like stratified negation