Datalog (1)

Goal: Learn *the* advanced optimization technique central to the (logical) optimization of queries on graphs

(In RDBMS, "home" for this material: *recursive queries*)

3 lectures:
- Learn Datalog, syntax, semantics, execution model(s)
  - a query language that leverages notation of first-order logic
  - concise & accessible
  - very commonly used in research papers
- Learn the Optimizing Transform
  - Magic-sets
  - a.k.a. sideways information passing
- Importance for graph queries will become self-evident

slide thanks: mostly Ullman, also Michael Lam

Datalog

Objectives, lecture 1:
- Introduce Datalog
  - Context, relation with/to
    - rule-based programming and logical inference
    - triggers
    - *business rules*, and supporting systems
  - recursion and querying graph-based data

slide thanks: mostly Ullman, also Michael Lam

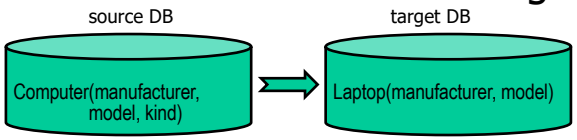
Logic As a Query Language

- If-then logic rules have been used in many systems.
  - Prolog
  - Triggers a.k.a. ECA rules: on **e**vent, if **c**ondition, **a**ction
  - Depending on your religion
    - AI, Knowledge-Base systems
      - Forward-chaining rules/production systems
      - Backward chaining rules, automatic proof systems
    - guarded command languages (do od)
      - Nod to Dijkstra and the formal methods community
      - Nondeterministic, fixed-point semantics

Logic As a Query Language

- Most important today: EII (Enterprise Information Integration)
  - Too many products to list
  - Special Mention: Logicblox (Datalog-based)
- Business logic/ workflow (BPEL, business process execution language)
  - IBM Websphere – ILOG rules
  - Redhat - DROOLS
  - … many others

View-based Information Integration




```
CREATE VIEW targetLaptop AS
        SELECT manufacturer, model
        FROM Computer
        WHERE Computer.kind = 'Laptop'
```

Datalog:

```
Laptop(manufacturer, model)
<- Computer(manufacturer, model, kind) AND kind = 'Laptop'
```


Logic As a Query Language

- Business logic/ workflow (BPEL, business process execution language)

Examples:

```
OvertimePay(person, date)
<- Role(person, title) AND OvertimeOK(title) AND Holiday(date)

VicePresidentSignatureRequired(purchase_id)
<- PurchaseOrder(puchase_id, amount) AND amount > $100,000
```


With Datalog Came Recursive Queries

- Nonrecursive subset of Datalog is equivalent to the core relational algebra.
- Recursive rules extend relational algebra --- have been used to add recursion to SQL-99.

Example Rule 1: a "happy drinker"

- Given
  - Frequents(drinker,bar) // defines drinker
  - Likes(drinker,beer), // what they like
  - Sells(bar,beer,price). // where sold, etc.
- Query who are the happy drinkers?

--> drinkers whose bars serve their favorite beer.

As a Datalog Rule

Happy(d) <- Frequents(d,bar) AND
Likes(d,beer) AND
Sells(bar,beer,p)

Example: Interpretation

Happy(d) <- Frequents(d,bar) AND
Likes(d,beer) AND Sells(bar,beer,p)

Distinguished variable

Nondistinguished variables

Interpretation: drinker d is happy ***if there exists*** a bar, a beer, and a price p such that d frequents the bar, likes the beer, and the bar sells the beer at price p.

Straight to the Critical Connection

Frequents	
Person	Bar
Juan	Dog & Duck
Dan	Crown & Anchor

Likes	
Person	Beer
Juan	Dos XX
Dan	Shiner

Sells		
Bar	Beer	Price
Dog & Duck	Dos XX	$5.00
Crown & Anchor	Dos XX	$4.00

Frequents(d,bar) AND Likes(d,beer) AND Sells(bar,beer,p)

```
select Person
from Frequents, Likes, Sells
where Frequents.Person = Likes.Person and
   Frequents.Bar = Sells.Bar and
   Likes.Beer = Sells.Beer
```

datalog, good: concise, looks familiar. bad: positional dependence

A Word on Notation

Frequents	
Person	Bar
Juan	Dog & Duck
Dan	Crown & Anchor

Likes	
Person	Beer
Juan	Dos XX
Dan	Shiner

Sells		
Bar	Beer	Price
Dog & Duck	Dos XX	$5.00
Crown & Anchor	Dos XX	$4.00

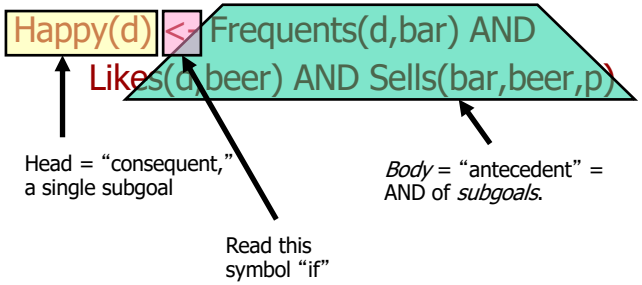
Frequents(d,bar) AND Likes(d,beer) AND Sells(bar,beer,p)

bad: positional dependence, d/Person, bar/Bar
don't care, use underscore: Frequents(_,bar)

Variables/Predicates

- Datalog (text): Variables, start with lower case, Predicates, upper case
- Datalog & Prolog (everywhere else):
  Predicates, start with lower case, Variables upper case

Anatomy of a Rule

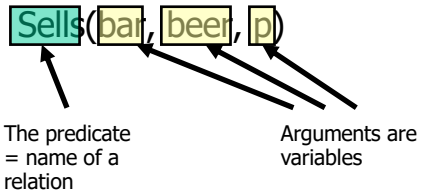


Subgoals Are Atoms

- An *atom* is a *predicate*, or relation name with variables or constants as arguments.
- The head is an atom; the body is the AND of one or more atoms.
- Convention: Predicates begin with a capital, variables begin with lower-case.

// At risk of confusion, your text repairs (changes) tradition: Prolog and Datalog, Predicates begin with lowercase letter, variables, an uppercase letter

Example: Atom



arity of a predicate: number of arguments in a predicate

Interpreting Rules

- A variable appearing in the head is called *distinguished* ; otherwise it is *nondistinguished*.
- Rule meaning: The head is true of the distinguished variables if there exist values of the nondistinguished variables that make all subgoals of the body true.

Arithmetic Subgoals

- In addition to relations as predicates, a predicate for a subgoal of the body can be an arithmetic comparison.
  - We write such subgoals in the usual way, e.g.: $x < y$.

Example: Arithmetic

- A beer is "cheap" if there are at least two bars that sell it for under $2.

```
Cheap(beer) <- Sells(bar1,beer,p1) AND
      Sells(bar2,beer,p2) AND p1 < 2.00
      AND p2 < 2.00 AND bar1 <> bar2
```


Negated Subgoals

- We may put NOT in front of a subgoal, to negate its meaning.
- Example:
- given additional relation
  HappyHour(bar,hours)
- unhappy drinker: none of his favorite bars have a happy hour

```
UnHappy(d) <- Frequents(d,bar) AND
              Not HappyHour(bar)
```


Relational Interpretation of Negation

- Set difference
  - A.k.a. negation as failure
  - Closed world assumption

Negated Subgoals

- unhappy drinker: none of his favorite bars have a happy hour

UnHappy(d) <- Frequents(d,bar) AND Not HappyHour(bar)

Frequents(d, bar) – HappyHour(bar)

Frequents: {(dan, 'crown anchor') (dan, 'jack allens')}
If
HappyHour: {'crown anchor'}, 'jack allens' remains, dan is happy
If
HappyHour: {'crown anchor', 'jack allens'}, Dan has no where to have happy hour, Dan is *bound* to d, Dan is unhappy

Closed and Open Worlds

- Given

unHappy(d) <- Frequents(d,bar) AND Not HappyHour(bar)
Frequents(d, bar) – HappyHour(bar)

Frequents: {(dan, 'crown anchor') (dan, 'jack allens')}
HappyHour: {'crown anchor', 'jack allens'},

Suppose, Dan also frequents Z'Tejas to drink. **But** it is not recorded in the database

Dan has a place to drink, he's happy,but database "doesn't know it"
Database is omniscient (knows everything): **Closed world**, set difference correctly computes NOT
Database does not know everything: **Open World**, can't logically conclude NOT

Datalog assumes a closed world

- that plus other properties make things computable.
- next property *safety*

Safe Rules

- A rule is *safe* if: // all the variables that appear in the // head, negated subgoal, or arithmetic subgoal, // appear in a nonnegated subgoal (can be bound)
  1. Each distinguished variable,
  2. Each variable in an arithmetic subgoal,
  3. Each variable in a negated subgoal,

  also appears in a nonnegated, relational subgoal.
- Datalog allows only safe rules.

Example: Unsafe Rules

- Each of the following is unsafe and not allowed:
  1. S(x) <- R(y)
  2. S(x) <- R(y) AND NOT R(x)
  3. S(x) <- R(y) AND x < y
- In each case, an infinity of *x*'s can satisfy the rule, even if *R* is a finite relation.

Datalog Programs

- A *Datalog program* is a collection of rules.
- In a program, predicates can be either
  1. EDB = *Extensional Database* = stored table. // base tables
  2. IDB = *Intensional Database* = relation defined by rules.
- Never both! No EDB in heads.

Evaluating Datalog Programs

- As long as there is no recursion, we can pick an order to evaluate the IDB predicates, so that all the predicates in the body of its rules have already been evaluated.
- If an IDB predicate has more than one rule, each rule contributes tuples to its relation.

Example: Datalog Program

- Using EDB Sells(bar, beer, price) and Beers(name, manf), find the manufacturers of beers Joe doesn't sell.

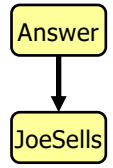
JoeSells(b) <- Sells('Joe''s Bar', b, p)
Answer(m) <- Beers(b,m)
AND NOT JoeSells(b)

Definition of Recursion

- Form a *dependency graph* whose nodes = IDB predicates.
- Arc *X* -> *Y* if and only if there is a rule with *X* in the head and *Y* in the body.
- Cycle = recursion; no cycle = no recursion.

Example: Dependency Graphs

JoeSells(b) <- Sells('Joe's Bar', b, p)
Answer(m) <- Beers(b,m)
AND NOT JoeSells(b)



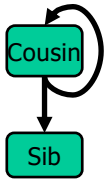
Nonrecursive

Recursive Example

- EDB: Par(c,p) = *p* is a parent of *c*.
- Generalized cousins: people with common ancestors one or more generations back:

Sib(x,y) <- Par(x,p) AND Par(y,p) AND x<>y
Cousin(x,y) <- Sib(x,y)
Cousin(x,y) <- Par(x,xp) AND Par(y,yp)
AND Cousin(xp,yp)

Example: Dependency Graphs



Sib(x,y) <- Par(x,p) AND Par(y,p) AND x<>y
Cousin(x,y) <- Sib(x,y)
Cousin(x,y) <- Par(x,xp) AND Par(y,yp)
AND Cousin(xp,yp)

Recursive

Expressive Power of Datalog

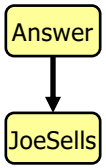
- Without recursion, Datalog can express all and only the queries of core relational algebra.
  - The same as SQL select-from-where, without aggregation and grouping.
- But with recursion, Datalog can express more than these languages.
- Yet still not Turing-complete.

Evaluation of Non-Recursive Datalog

- Topologically sort the dependency graph.
- Evaluate rules in sorted order

Example: Dependency Graphs

JoeSells(b) <- Sells('Joe's Bar', b, p)
Answer(m) <- Beers(b,m)
AND NOT JoeSells(b)



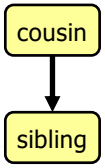
Nonrecursive

Consider Subset of Cousin Program

```
(1) sibling(X,Y) :- parent(X,Z) & parent(Y,Z) & X≠Y.
(2) cousin(X,Y) :- parent(X,Xp) & parent(Y,Yp) &
                   sibling(Xp,Yp).
```

Consider Subset of Cousin Program

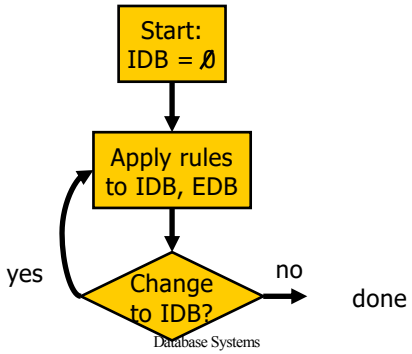
```
(1) sibling(X,Y) :- parent(X,Z) & parent(Y,Z) & X≠Y.
(2) cousin(X,Y) :- parent(X,Xp) & parent(Y,Yp) &
                   sibling(Xp,Yp).
```



Evaluating Recursive Rules

- The following works when there is no negation:
  1. Start by assuming all IDB relations are empty.
  2. Repeatedly evaluate the rules using the EDB and the previous IDB, to get a new IDB.
  3. End when no change to IDB. (fixed-point)

The "Naïve" Evaluation Algorithm



Example: Evaluation of Cousin

- We' ll proceed in rounds to infer Sib facts (red) and Cousin facts (green).
- Remember the rules:

Sib(x,y) <- Par(x,p) AND Par(y,p) AND x<>y
Cousin(x,y) <- Sib(x,y)
Cousin(x,y) <- Par(x,xp) AND Par(y,yp)
AND Cousin(xp,yp)

Seminaive Evaluation

- Since the EDB never changes, on each round we only get new IDB tuples if we use at least one IDB tuple that was obtained on the previous round.
- Saves work; lets us avoid rediscovering *most* known facts.
  - A fact could still be derived in a second way.

Incremental Maintenance of a View

$$(R \cup S) \,|X|\, T = (R \,|X|\, T) \cup (S \,|X|\, T)$$

So, if View, V = (S |X| T)

- Insert Into S, rows R, i.e. S' = (R ∪ S)
- New view contents, V', V' = S' |X| T

V' = (R ∪ S)|X| T
= (R |X| T) ∪ (S |X| T)
= (R |X| T) ∪ V

Par Data: Parent Above Child

```
(1) sibling(X,Y) :- parent(X,Z) & parent(Y,Z) & X≠Y.
(2) cousin(X,Y) :- parent(X,Xp) & parent(Y,Yp) &
                   sibling(Xp,Yp).
(3) cousin(X,Y) :- parent(X,Xp) & parent(Y,Yp) &
                   cousin(Xp,Yp).
```

Red: sibling rows
Green: cousin rows