CPSC 304
Introduction to Database Systems

Datalog & Deductive Databases

Textbook Reference

Database Management Systems: Sections 24.1 – 24.4

Databases: The Continuing Saga

When last we left databases…

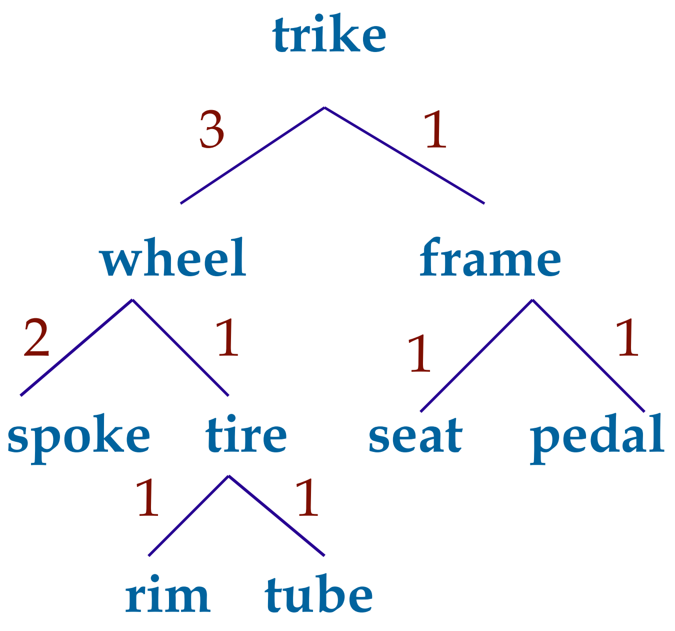
- We had decided they were great things
- We knew how to conceptually model them in ER diagrams
- We knew how to logically model them in the relational model
- We knew how to normalize them
- We learned relational algebra
- We learned SQL

Let’s talk about our final database query language – Datalog!

Learning Goals

- Given a set of tuples (an input relation) and rules, compute the output relation for a Datalog program.
- Write Datalog programs to query an input relation.
- Explain why we want to extend query languages with recursive queries. Provide good examples of such queries.
- Explain the importance of safe queries, and what makes a Datalog query safe.

Motivation



part	subpart	qty
trike	wheel	3
trike	frame	1
frame	seat	1
frame	pedal	1
wheel	spoke	2
wheel	tire	1
tire	rim	1
tire	tube	1

Try to write a relational algebra query to find all of the components required for a trike

Datalog

- Based on logic notation (Prolog)
- Can express queries that are not expressible in relational algebra or standard SQL (recursion).
- Uses sets (like RA, unlike SQL)
- Cleaner → convenient for analysis

A nice and easy example to start

From a query perspective: ask a query and get answers.

From a logical perspective: use facts to derive new facts.

Tuples/Initial facts:

Parent(“Dee”, “Jan”)

Parent(“Jan”, “Jamie”)

Parent(“Dee”, “Wally”)

Parent(“Wally”, “Jean”)

Query:

Grandparent(A,C) :- Parent(A,B), Parent(B,C)

Answer/New facts:

Grandparent(“Dee”, “Jamie”)

Grandparent(“Dee”, “Jean”)

Predicates and Atoms

- Relations are represented by predicates
- Tuples are represented by atoms.

  Parent(“Dee”, “Jan”)

Arithmetic comparison atoms:

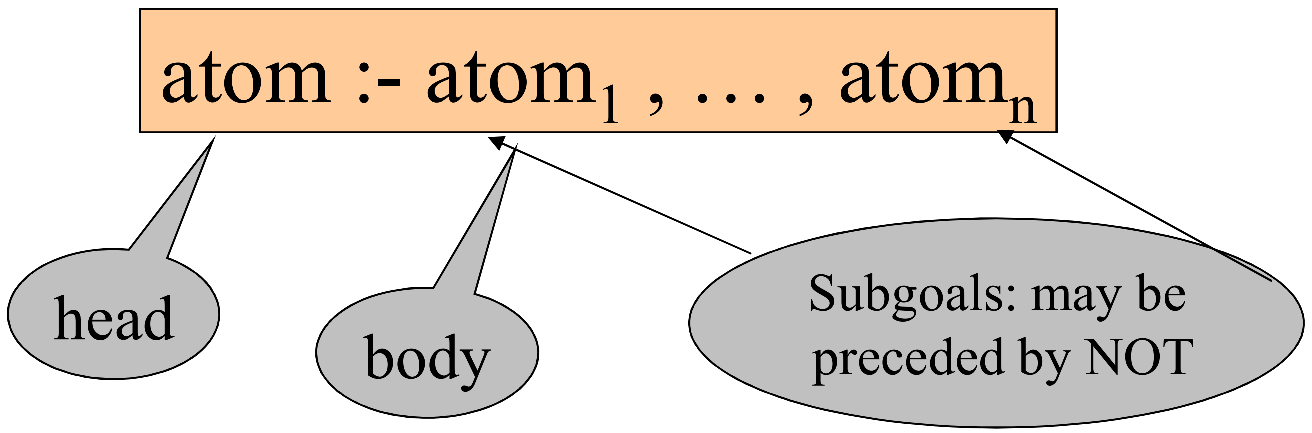
X < 100, X+Y+5 > Z/2, X <> 42

- Negated atoms:

  NOT Parent(“Dee”, “Jean”)

Datalog Definitions

- A Datalog rule:



$$\text{atom :- atom}_1 \text{ , … , atom}_n$$

- E.g.: Grandparent(A,C) :- Parent(A,B), Parent(B,C).
- A comma between the atoms means “and” (sometimes you’ll see this as “&”)
- Read the rule as “if we know body, then we know head”
- You may also see head ← body, e.g., Grandparent(A,C)← Parent(A,B), Parent(B,C)
- Datalog program = a collection of rules

A single rule can express exactly select-project-join queries.

The Meaning of Datalog Rules

```
Parent("Dee", "Jan").
Parent("Jan", "Jamie").
Parent("Dee", "Wally").
Parent("Wally", "Jean").
Grandparent(A,C) :- Parent(A,B), Parent(B,C).
```

```
Grandparent("Dee", "Jamie")
Grandparent("Dee", "Jean")
```

Consider every assignment from the variables in the body to the constants in the database. (same variable name means require the same value)

If each atom in the body is in the database, then the tuple for the head is in the result.

Running example

Product (pid, name, price, category, maker-cid)
Purchase (buyer-sin, seller-sin, store, pid)
Company (cid, name, stock price, country)
Person(sin, name, phone number, city)

Projection

Product (<u>pid</u>, name, price, category, maker-cid)
Purchase (buyer-sin, seller-sin, store, pid)
Company (<u>cid</u>, name, stock price, country)
Person(<u>sin</u>, name, phone number, city)

Projection is performed by the variables in the head of the query:

- Find the name of all products:

  RA: π_{name}(Product)

  Datalog: Ans(N):-Product(P,N,PR,C,M)

Projection practice

Product (<u>pid</u>, name, price, category, maker-cid)
Purchase (buyer-sin, seller-sin, store, pid)
Company (<u>cid</u>, name, stock price, country)
Person(<u>sin</u>, name, phone number, city)

Find the countries of all the companies

Ans1(Co):- Company (C, N, S, Co)

Note: make sure C $\neq$Co

Selection

Product (pid, name, price, category, maker-cid)
Purchase (buyer-sin, seller-sin, store, pid)
Company (cid, name, stock price, country)
Person(sin, name, phone number, city)

Selection is performed by either using the same variable, a constant, or adding an arithmetic comparison:

- Find all purchases with the same buyer and seller:
  RA: $\sigma_{\text{buyer-sin = seller-sin}}$(Purchase)
  Datalog: Ans1(B,B,S,P):-Purchase(B,B,S,P)

- Find all Canadian companies:
  RA: $\sigma_{\text{country='Canada'}}$(Company)
  Datalog: Ans2(C,N,S, 'Canada'):-Company(C,N,S, 'Canada')
  Alternate option: Ans2(C,N,S, Co):-Company(C,N,S, Co),
  Co = 'Canada'

Selection & Projection and Joins

Product (pid, name, price, category, maker-cid)
Purchase (buyer-sin, seller-sin, store, pid)
Company (cid, name, stock price, country)
Person(sin, name, phone number, city)

Joins are performed by using the same variable in different relations

- Find store names where Fred bought something:
  RA: $\pi_{store}(\sigma_{name=\text{“Fred”}}(Person) \bowtie_{sin=buyer\text{-}sin} Purchase)$
- Datalog: S(N) :- Person(S, “Fred”,T,C),
  Purchase(S,L,N,P)

Anonymous Variables

Product (pid, name, price, category, maker-cid)
Purchase (buyer-sin, seller-sin, store, pid)
Company (cid, name, stock price, country)
Person(sin, name, phone number, city)

Find names of people who bought from “Gizmo Store”

E.g.:

```
  Ans4(N)  :- Person(S, N, _, _), Purchase (S, _,“Gizmo Store”, _)
```

Each _ means a fresh, new variable
Very useful: makes Datalog even easier to read

Anonymous Variables

Our ongoing schema:

Product (<u>pid</u>, name, price, category, maker-cid)

Purchase (buyer-sin, seller-sin, store, pid)

Company (<u>cid</u>, name, stock price, country)

Person(<u>sin</u>, name, phone number, city)

Write queries in relational algebra and Datalog to find the **names of all products**

Relational algebra: π_{name}(Product)

Datalog: q(n):-Product(_, n, _, _, _)

Anonymous Variables

Our ongoing schema:

Product (<u>pid</u>, name, price, category, maker-cid)

Purchase (buyer-sin, seller-sin, store, pid)

Company (<u>cid</u>, name, stock price, country)

Person(<u>sin</u>, name, phone number, city)

Write queries in relational algebra and Datalog to find the **names of people who have bought products from themselves**

RA: $\pi_{\text{name}}((\sigma_{\text{buyer-sin=seller-sin}}\text{Purchase}) \bowtie_{\text{seller-sin = sin}} \text{Person})$

Datalog: q(n):-Person(s,n,_,_), Purchase(s, s,_, _)

Multiple Datalog Rules

Product (pid, name, price, category, maker-cid)
Purchase (buyer-sin, seller-sin, store, pid)
Company (cid, name, stock price, country)
Person(sin, name, phone number, city)

- Find names of people that are either buyers or sellers:

```
A(N) :- Person(S,N,A,B), Purchase(S,C,D,E)
A(N) :- Person(S,N,A,B), Purchase(C,S,D,E)
```

- Multiple rules correspond to union

Great! But surely there's more…

Our ongoing schema:

Product (<u>pid</u>, name, price, category, maker-cid)

Purchase (buyer-sin, seller-sin, store, pid)

Company (<u>cid</u>, name, stock price, country)

Person(<u>sin</u>, name, phone number, city)

Write a query in relational algebra to find SINs of people who have not made a purchase

$\pi_{sin}(\text{Person}) - \pi_{buyer\text{-}sin}(\text{Purchase})$

Negation

Find people who live in Vancouver but have not bought anything at “The Bay”

```
VancouverAntiBay(buyer,seller,product,store) :-
    Person(buyer, name, phone,“Vancouver”),
    Purchase(buyer, seller, store, product),
   not  Purchase(buyer, seller, “The Bay”, product)
```

The NOT in Datalog means “there exists no”

You may also see “NOT” written as “┐”

Defining Queries for reuse: Views

```
VancouverAntiBay(Buyer,Seller,Product,Store) :-
    Person(Buyer, "Vancouver", Phone),
    Purchase(Buyer, Seller, Product, Store),
    not  Purchase(Buyer, Seller, Product, "The Bay")

Ans6(Buyer) :- VancouverAntiBay(Buyer, "Joe", Pro, Store)
Ans6(Buyer) :- VancouverAntiBay(Buyer, Sell, Prod, Store),
               Product(Prod, Price, Cat, Maker)
               Company(Maker, Sp, Country), Sp > 50.
```

What is returned by Ans6?

Buyers from Vancouver that have never purchased anything from "The Bay" that have either bought from Joe or products that are from companies with SP> 50

Rule safety

- Every variable in the head of a rule must also appear in the body.

PriceParts (Part, Price) :- Assembly(Part, Subpart, Qty) , Qty> 2.

Can generate infinite new facts (what is price)?

- Every variable must appear in a relation

Ans(Id) :- Product(Id,Name,Price,Category,Cid), Id < Stock_price

What is the value of stock_price?

- Every variable in the head of the rule must appear in some positive relation occurrence in the body and every variable a negated form must appear positively in the body

Ans(Sin):- NOT Person(Sin, 'Joe', Ph, City)

Sin, Ph, and City are unsafe

Division in Datalog

Assume schema

Cust($\underline{\text{cid}}$,cname,rating,salary)

Order($\underline{\text{iid,cid,day}}$,qty)

Query: find items (iid) that are ordered by every customer

First: write the query in Relational Algebra

$\pi_{iid,cid}(order)/\pi_{cid}(cust)$

Reminder: building up division subtract off disqualified answers in RA

$A = R$

Sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

$B2 = S$

Pno
P2
P4

$\pi_X(R)$

Sno
S1
S2
S3
S4

All possible values given R

$\pi_X(R) \times S$

Sno	Pno
S1	P2
S1	P4
S2	P2
S2	P4
S3	P2
S3	P4
S4	P2
S4	P4

Values needed for $\pi_X(R)$

$\pi_X(R) \times S - R$

Sno	Pno
S2	P4
S3	P4

Missing to have $\pi_X(R)$

$$\pi_X(R) - \pi_X(\pi_X(R) \times S - R) = A/B2$$

A/B2 =

Sno
S1
S4

Answers not disqualified

What does this look like in Datalog?

```
Witness(I,C):-Order(I,C,_,_)
Bad(I)          :- Cust(_,C,_,_), Order(I,_,_,_), ¬ Witness(I,C)
Good(I)        :- Order(I,C, _, _), ¬ Bad(I)
```

Technically, you don’t need witness, can just use another “order”. It’s just cleaner to do it this way.

- Witness finds all items that have been ordered
- Bad finds all items that have not been ordered by some customer
- Good: finds all items for that have not been not ordered by some customer (i.e., all items that have been ordered by all customers)

More generally, in the simplest case

Relations: A(A1, B1, …), B(B1, …)

Witness(a1,b1):-A(a1,b1,…)

Bad(a1) :- A(a1,…), B(b1,…), ┐Witness(a1,b1)

Good(a1) :- A(a1,…), ┐Bad(a1)

Returning to our previous example

A(Sno, Pno)

Sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B(Pno)

Pno
P2
P4

Witness(a1,b1):-
A(a1, b1)

a1	b1
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

A/B =

a1
S1
S4

Bad-no-π(a1,b1):-
A(a1,…), B(b1,…)
┐Witness(a1,b1)

a1	b1
S2	P4
S3	P4

Bad(a1):-
A(a1,…), B(b1,…)
┐Witness(a1,b1)

Bad

a1
S2
S3

Good(a1):-A(a1,…),
┐Bad(a1)

Taking it to the next level

Say you're planning a beach vacation

And you wanted to find if it's possible to get from YVR to OGG (that's on Maui)

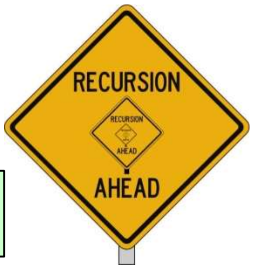
Your available information:

Flight(airline,num,origin,destination)

Now what?

A more general Example: Transitive Closure

Not tested on exam



Suppose we represent a graph w/ relation *Edge(X,Y):*
Edge(a,b), Edge (a,c), Edge(b,d), Edge(c,d), Edge(d,e)

$$\begin{tikzcd} & b \arrow[dr] & & \\ a \arrow[ur] \arrow[dr] & & d \arrow[r] & e \\ & c \arrow[ur] & & \end{tikzcd}$$

How can I express the query: *Find all paths*

Path(X, Y) :- Edge(X, Y).
Path(X, Y) :- Path(X, Z), Path(Z, Y).

Not tested on exam

Evaluating Recursive Queries

Path(X, Y) :- Edge(X, Y).
Path(X, Y) :- Path(X, Z), Path(Z, Y).

Semantics: evaluate the rules until a *fixed point:*

Iteration #0: Edge: {(a,b), (a,c), (b,d), (c,d), (d,e)}
Path: {}

Iteration #1: Path: {(a,b), (a,c), (b,d), (c,d), (d,e)}

Iteration #2: Path gets the new tuples: (a,d), (b,e), (c,e)
Path: {(a,b), (a,c), (b,d), (c,d), (d,e), (a, d), (b,e), (c, e)}

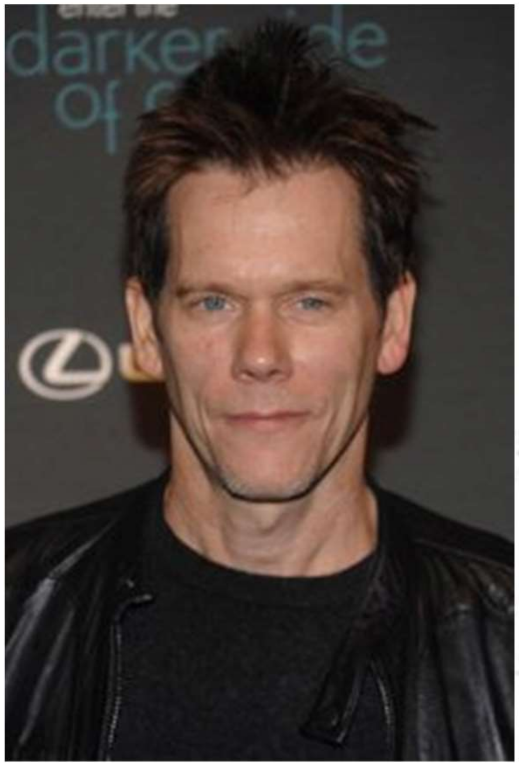
Iteration #3: Path gets the new tuple: (a,e)
Path: {(a,b), (a,c), (b,d), (c,d), (d,e), (a, d), (b,e), (c, e), (a,e)}

Iteration #4: Nothing changes → Stop.

Note: # of iterations depends on the data. Cannot be anticipated by only looking at the query!

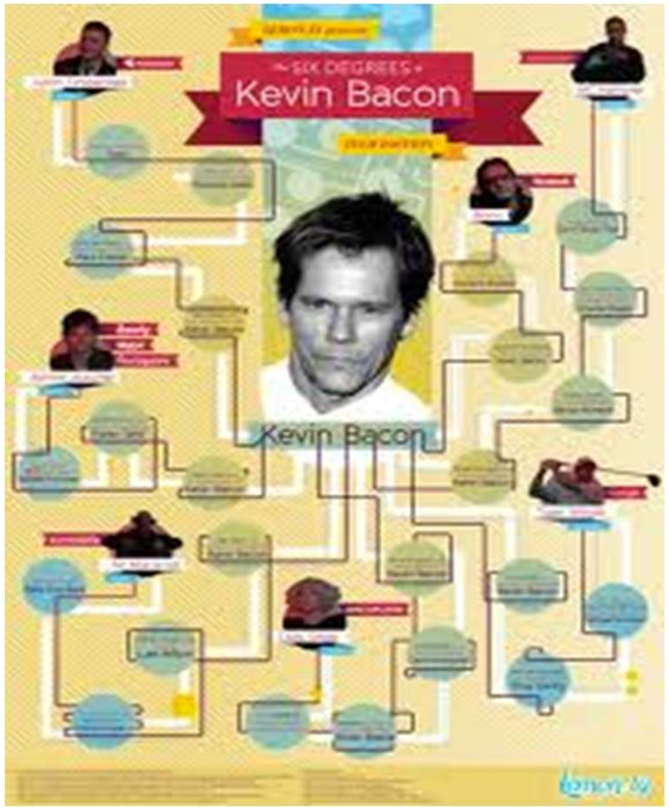
A fun Example

Kevin Bacon



6 degrees of separation

6 degrees of Kevin Bacon



More examples

- Given:
  Movie(id, title)
  Actor(id, name)
  Role(movie-id, actor-id, character)
- Find names of actors who have “Bacon numbers” (assume there’s only one “Kevin Bacon”)

```
CoStars(Aid,Bid):-Role(Mid,Aid,_), Role(Mid,Bid,_)
CoStars(Aid,Bid):- CoStars(Aid,Cid), CoStars(Cid,Bid)
Bacon_N(B):-Actor(Aid, "Kevin Bacon"), CoStars(Aid,Bid),
                   Actor(Bid,B)
```

Not tested on exam

Recursive SQL? Sometimes…

Given: Assembly(Part, Subpart, Quantity)
Find: all of the components required for a trike

Datalog:

```
Comp(Part, Subpt) :- Assembly(Part, Subpt, Qty).
Comp(Part, Subpt) :- Assembly(Part, Part2, Qty), Comp(Part2, Subpt).
```

SQL:

```
WITH RECURSIVE Comp(Part, Subpt) AS
    (SELECT A1.Part, A1.Subpt FROM Assembly A1)
  UNION
    (SELECT A2.Part, C1.Subpt
    FROM Assembly A2, Comp C1
    WHERE A2.Subpt=C1.Part)

SELECT Subpart FROM Comp C2
WHERE Part = ‘trike’
```

On the off chance that you have the book… Skip the stuff on Magic Sets

- That’s Datalog
- It’s simple
- It’s based on logic
- It’s easy to see the join patterns (especially with anonymous variables)

Learning Goals Revisited

- Given a set of tuples (an input relation) and rules, compute the output relation for a Datalog program.
- Write Datalog programs to query an input relation.
- Explain why we want to extend query languages with recursive queries. Provide good examples of such queries.
- Explain the importance of safe queries, and what makes a Datalog query safe.