Datalog (2)
Evaluating Datalog Programs

Consider

```
CREATE VIEW A AS
        SELECT B.b1 AS a1, B.b2 AS a2
        FROM B
```

** no where clause on purpose

Consider

```
CREATE VIEW A AS
        SELECT B.b1 AS a1, B.b2 AS a2
        FROM B
```

As a Datalog rule:

A(a1, a2) ← B(a1, a2) ** remember positions, not names

Consider

```
CREATE VIEW A AS
        SELECT B.b1 AS a1, B.b2 AS a2  FROM B

CREATE VIEW B AS
        SELECT C.c1 AS b1, C.c2 AS b2  FROM C

CREATE VIEW C AS
        SELECT D.d1 AS c1, D.d2  FROM D


CREATE TABLE D (d1 integer, d2 integer);
```

If D contains (8, 16)

```
CREATE VIEW A AS
        SELECT B.b1 AS a1, B.b2 AS a2  FROM B

CREATE VIEW B AS
        SELECT C.c1 AS b1, C.c2 AS b2  FROM C

CREATE VIEW C AS
        SELECT D.d1 AS c1, D.d2  FROM D

CREATE TABLE D (d1 integer, d2 integer);
```

What does SELECT * FROM A
Return?

Make a plan, compute: A = {(8, 16)}

What does SELECT * FROM A
Return?

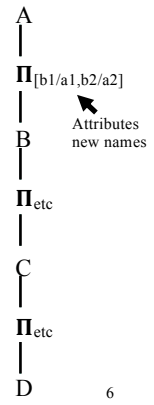
```
CREATE VIEW A AS
        SELECT B.b1 AS a1, B.b2 AS a2  FROM B

CREATE VIEW B AS
        SELECT C.c1 AS b1, C.c2 AS b2  FROM C

CREATE VIEW C AS
        SELECT D.d1 AS c1, D.d2  FROM D

CREATE TABLE D (d1 integer, d2 integer);
```

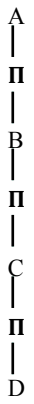
A *top-down* evaluation



As a Datalog program

A(a1, a2) ← B(a1, a2)
B(b1, b2) ← C(b1, b2)
C(c1, c2) ← D(c1, c2)

D = {(8, 16)}



Then, Suppose:

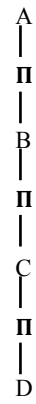
INSERT INTO D VALUES(32, 64)

As a Datalog program

A(a1, a2) ← B(a1, a2)
B(b1, b2) ← C(b1, b2)
C(c1, c2) ← D(c1, c2)

D = {(8, 16), (32, 64)}

We can re-execute the plan



But what if

```
CREATE TABLE D (d1 integer, d2 integer);
INSERT INTO D VALUES(8,16);

CREATE MATERIALIZED VIEW C AS
        SELECT D.d1 AS c1, D.d2  FROM D

CREATE MATERIALIZED VIEW B AS
        SELECT C.c1 AS b1, C.c2 AS b2  FROM C

CREATE MATERIALIZED VIEW A AS
        SELECT B.b1 AS a1, B.b2 AS a2  FROM B
```


And Then
INSERT INTO D VALUES(32, 64)

D = {(8, 16)} ∪ (32, 64)

Incrementally Maintain Materialized View

From past lecture, but incrementally maintained view had a join

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

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

$$V' = (R \cup S) \,|X|\, T$$
$$= (R \,|X|\, T) \cup (S \,|X|\, T)$$
$$= (R \,|X|\, T) \cup V$$

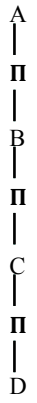
So:

If D = {(8, 16)}

C(c1, c2) ← D(c1, c2)
B(b1, b2) ← C(b1, b2)
A(a1, a2) ← B(a1, a2)

C = {(8, 16)}
B = {(8, 16)}
A = {(8, 16)}

D = {(8, 16), (32, 64)}



Further:

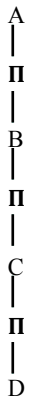
And If D = {(8, 16)} U {(32, 64)}

C(c1, c2) ← D(c1, c2)
B(b1, b2) ← C(b1, b2)
A(a1, a2) ← B(a1, a2)

C = {(8, 16)} U {(32, 64)}
B = {(8, 16)} U {(32, 64)}
A = {(8, 16)} U {(32, 64)}

By using incremental view maintenance method:

(32, 64) was added to C, but (8, 16) not recomputed
Then
(32, 64) was added to B, but (8, 16) not recomputed
…
A *bottom-up* evaluation



In Datalog, this is called *semi-naïve evaluation*

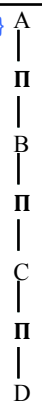
And If D = {(8, 16)} U {(32, 64)}

C(c1, c2) ← D(c1, c2)
B(b1, b2) ← C(b1, b2)
A(a1, a2) ← B(a1, a2)

C = {(8, 16)} U {(32, 64)}
B = {(8, 16)} U {(32, 64)}
A = {(8, 16)} U {(32, 64)}

Next steps, engage with:

- Only think about Datalog, and not to trivial programs
- Deal with recursion
- Deal with recursion and negation *(need stratification)*



Example: Datalog Program

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

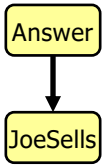
JoeSells(b) <- Sells('Joe's Bar', b, p)
Answer(m) <- Beers(b,m)
AND NOT JoeSells(b)

Dependency Graph

- 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.

Example: Dependency Graphs

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



Nonrecursive

Evaluating Non-recursive Datalog Programs

- Build the IDB dependency graph
- Topologically sort the graph
- Evaluate the IDB predicates,
  - Topological sort →
    - 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.

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.

Expressive Power of Datalog

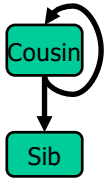
- Without recursion, Datalog can express all and only the queries of core relational algebra.
  - The same as SQL select-from-where, without aggregation and grouping.
- But with recursion, Datalog can express more than these languages.
- Yet still not Turing-complete.

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)

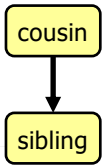
Recursive

Consider Subset of Cousin Program

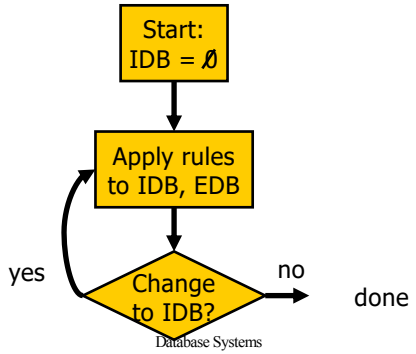
```
(1) sibling(X,Y) :- parent(X,Z) & parent(Y,Z) & X≠Y.
(2) cousin(X,Y) :- parent(X,Xp) & parent(Y,Yp) &
                   sibling(Xp,Yp).
```


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).
```



The "Naïve" Evaluation Algorithm



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)

Example: Evaluation of Cousin

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

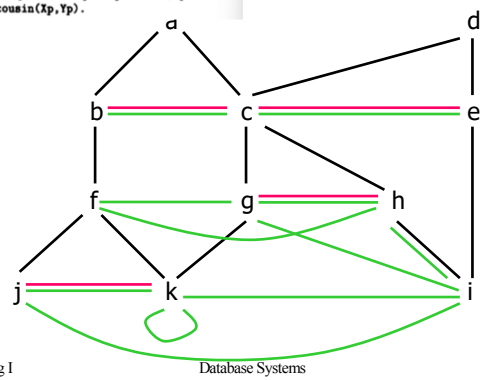
Sib(x,y) <- Par(x,p) AND Par(y,p) AND x<>y
Cousin(x,y) <- Sib(x,y)
Cousin(x,y) <- Par(x,xp) AND Par(y,yp)
AND Cousin(xp,yp)

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

Round 1
Round 2
Round 3
Round 4



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, (a different chain of rules)

Recursion Plus Negation

- "Naïve" evaluation doesn't work when there are negated subgoals.
- In fact, negation wrapped in a recursion makes no sense in general.
- Even when recursion and negation are separate, we can have ambiguity about the correct IDB relations.

Problematic Recursive Negation

P(x) <- Q(x) AND NOT P(x)
EDB: Q(1), Q(2)

Initial: P = { }
Round 1: P = {(1), (2)}
Round 2: P = { }
Round 3: P = {(1), (2)}, etc., etc. …

Stratified Negation

- Stratification is a constraint usually placed on Datalog with recursion and negation.
- It rules out negation wrapped inside recursion.
- Gives the sensible IDB relations when negation and recursion are separate.

Strata

- Intuitively, the *stratum* of an IDB predicate *P* is the maximum number of negations that can be applied to an IDB predicate used in evaluating *P*.
- Stratified negation = "finite strata."
- Notice in P(x) <- Q(x) AND NOT P(x), we can negate *P* an infinite number of times deriving P(x).

Stratum Graph

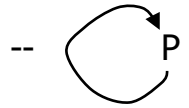
- To formalize strata use the *stratum graph* :
  - Nodes = IDB predicates.
  - Arc *A* ->*B* if predicate *A* depends on *B*.
  - Label this arc "–" if the *B* subgoal is negated.

Stratified Negation Definition

- The *stratum* of a node (predicate) is the maximum number of – arcs on a path leading from that node.
- A Datalog program is *stratified* if all its IDB predicates have finite strata.

The Problematic Example

P(x) <- Q(x) AND NOT P(x)



Recursion cycles through a negation

Another Example

- EDB = Source(x), Target(x), Arc(x,y).
- Rules for "targets not reached from any source":

Reach(x) <- Source(x)

Reach(x) <- Reach(y) AND Arc(y,x)

NoReach(x) <- Target(x)
AND NOT Reach(x)

** Reachability

- Reachability, as a fundamental capability of graph databases

→A core capability used to promote them

** Reachability

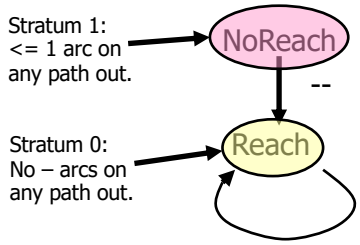
- Reachability, as a fundamental capability of graph databases

→A core capability used to promote them

- Covid-19 contact mapping:
  - Graph data:
    - vertex is a person,
    - Edge connects two people if they were in physical proximity and, the edge is labeled with the property (key) of the time and date of the contact (value). E.g. (contact, "April 5, 12:07pm")
  - Who were possible sources of John's Covid-19?
    - All vertices reachable from John in the last 7 days.
  - Who is the most likely source?
    - What is the shortest path to symptomatic person?

- Each person is represented by vertex.
- An edge exists from p1 to p2, if

The Stratum Graph



Models

- A *model* is a choice of IDB relations that, with the given EDB relations makes all rules true regardless of what values are substituted for the variables.
  - Remember: a rule is true whenever its body is false.
  - But if the body is true, then the head must be true as well.

Minimal Models

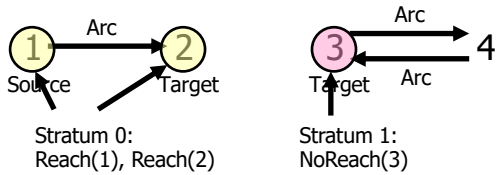
- When there is no negation, a Datalog program has a unique minimal model (one that does not contain any other model).
- But with negation, there can be several minimal models.
- The stratified model is the one that "makes sense."

The Stratified Model

- When the Datalog program is stratified, we can evaluate IDB predicates lowest-stratum-first.
- Once evaluated, treat it as EDB for higher strata.

Example: Multiple Models --- (1)

Reach(x) <- Source(x)
Reach(x) <- Reach(y) AND Arc(y,x)
NoReach(x) <- Target(x) AND NOT Reach(x)



Stratum 0:
Reach(1), Reach(2)

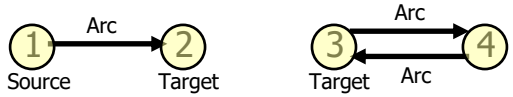
Stratum 1:
NoReach(3)

Assumption

- When the logic is stratified, the stratified model is the one that "makes sense."
- This principle is used in SQL-99 recursion --- the stratified model is defined to be the correct query result.

Example: Multiple Models --- (2)

Reach(x) <- Source(x)
Reach(x) <- Reach(y) AND Arc(y,x)
NoReach(x) <- Target(x) AND NOT Reach(x)



Another model! Reach(1), Reach(2), Reach(3), Reach(4); NoReach is empty.