CompSci 516
Database Systems

Lecture 21
Recursive Query Evaluation and Datalog

Instructor: Sudeepa Roy

Annoucements

- Office hour (Sudeepa) until 12 noon today
  - send me an email if this does not work and you want to meet
- HW3 due next Monday

Where are we now?

We learnt

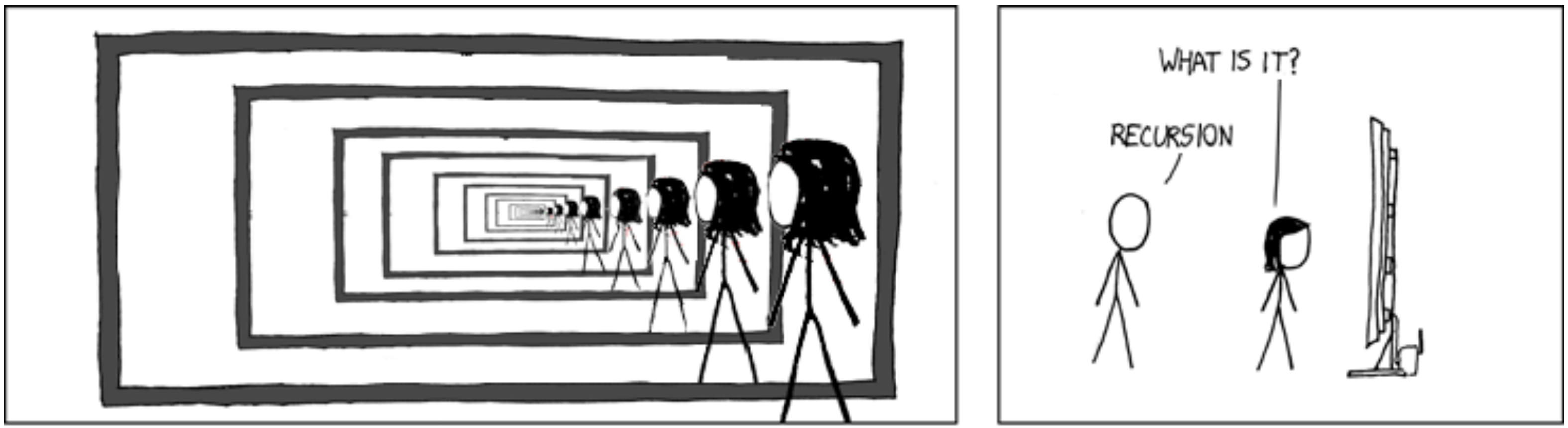
- ✓ Relational Model and Query Languages
  - ✓ SQL, RA, RC
  - ✓ Postgres (DBMS)
    - HW1
- ✓ Database Normalization
- ✓ DBMS Internals
  - ✓ Storage
  - ✓ Indexing
  - ✓ Query Evaluation
  - ✓ Operator Algorithms
  - ✓ External sort
  - ✓ Query Optimization
- ✓ Map-reduce and spark
  - HW2

- Transactions
  - Basic concepts
  - Concurrency control
  - Recovery
- Distributed DBMS
- NOSQL
- Parallel DBMS

Today

- Semantic of recursion in databases
- Datalog
  - for recursion in database queries
- Semi-naïve evaluation using
  - Incremental View Maintenance (IVM)
  - What is a view

Recursion!

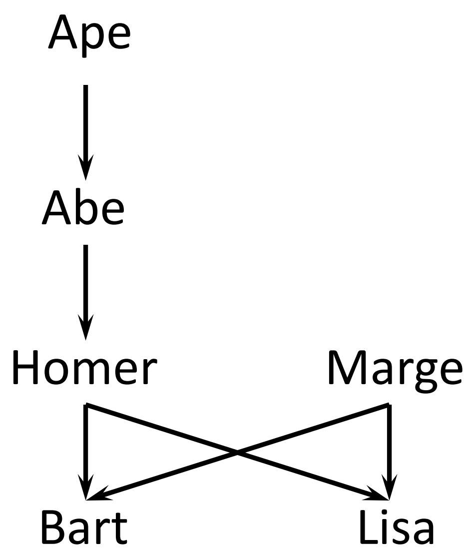


http://xkcdsw.com/1105

A motivating example

Parent (*parent*, *child*)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe



- Example: find Bart's ancestors
- "Ancestor" has a recursive definition
  - X is Y's ancestor if
    - X is Y's parent, or
    - X is Z's ancestor and Z is Y's ancestor

Recursion in SQL

- SQL2 had no recursion
  - You can find Bart's parents, grandparents, great grandparents, etc.

```
SELECT p1.parent AS grandparent
FROM Parent p1, Parent p2
WHERE p1.child = p2.parent
AND p2.child = 'Bart';
```

  - But you cannot find all his ancestors with a single query

Recursion in Databases

- Consider a graph G(V, E). Can you find out all “ancestor” vertices that can reach “x” using Relational Algebra/Calculus?
- NO! – ANCESTOR cannot be defined using a finite union of select-project-join queries (conjunctive queries)
- No RA/RC expressions can express ANCESTOR or REACHABILITY (TRANSITIVE CLOSURE) (Aho-Ullman, 1979)
- A limitation of RA/RC in expressing recursive queries

Recursion in Databases

- What can we do to overcome the limitation?

1. Embed SQL in a high level language supporting recursion
   - (-) destroys the high level declarative characteristic of SQL
2. Augment RC with a high level declarative mechanism for recursion
   - Datalog (Chandra-Harel, 1982)

- SQL:1999 (SQL3) and later versions support “linear Datalog”

Brief History of Datalog

- Motivated by Prolog – started back in 80's – then quiet for a long time
- A long argument in the Database community whether recursion should be supported in query languages
  - *"No practical applications of recursive query theory ... have been found to date"*—Michael Stonebraker, 1998
    Readings in Database Systems, 3rd Edition Stonebraker and Hellerstein, eds.
  - Recent work by Hellerstein et al. on Datalog-extensions to build networking protocols and distributed systems. [Link]

Datalog is resurging!

- Number of papers and tutorials in DB conferences

- Applications in
  - data integration, declarative networking, program analysis, information extraction, network monitoring, security, and cloud computing

- Systems supporting datalog in both academia and industry:
  - Lixto (information extraction)
  - LogicBlox (enterprise decision automation)
  - Semmle (program analysis)
  - BOOM/Dedalus (Berlekey)
  - Coral
  - LDL++

Reading Material: Datalog

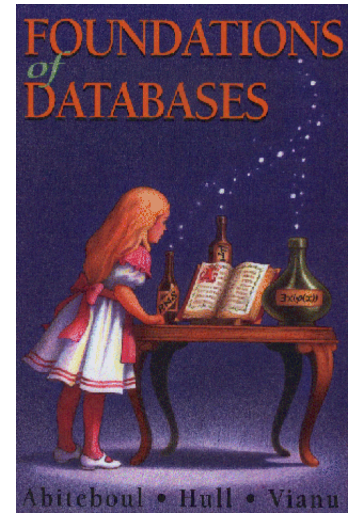
Optional:

1. The datalog chapters in the "Alice Book"

Foundations of Databases

Abiteboul-Hull-Vianu

Available online: http://webdam.inria.fr/Alice/



2. Datalog tutorial

SIGMOD 2011

"Datalog and Emerging Applications: An Interactive Tutorial"

Acknowledgement:
Some of the following slides have been borrowed from slides by Prof. Jun Yang

Recursive Query in SQL

Recursion in SQL

- SQL2 had no recursion
- SQL3 introduces recursion
  - WITH clause
  - Implemented in PostgreSQL (common table expressions)

Ancestor query in SQL3

Define a relation recursively

```
WITH RECURSIVE
Ancestor(anc, desc) AS
(
 (SELECT parent, child FROM Parent)
  UNION
 (SELECT a1.anc, a2.desc
     FROM Ancestor a1, Ancestor a2
     WHERE a1.desc = a2.anc)
)
SELECT anc
FROM Ancestor
WHERE desc = 'Bart';
```

base case

recursion step

Query using the relation defined in WITH clause

Fixed point of a function

- If $f: T \to T$ is a function from a type T to itself, a fixed point of f is a value x such that $f(x) = x$
- Example: What is the fixed point of $f(x) = x/2$?
  - 0, because $f(0) = 0/2 = 0$

To compute fixed point of a function f

- Start with a "seed": $x \leftarrow x_0$
- Compute $f(x)$
  - If $f(x) = x$, stop; x is fixed point of f
  - Otherwise, $x \leftarrow f(x)$; repeat

- Example: compute the fixed point of $f(x) = x/2$
  - With seed 1: 1, 1/2, 1/4, 1/8, 1/16, … → 0

☞Doesn't always work, but happens to work for us!

Fixed point of a query

- A query q is just a function that maps an input table to an output table, so a fixed point of q is a table T such that $q(T) = T$

To compute fixed point of q

- Start with an empty table: $T \leftarrow \emptyset$
- Evaluate q over T
  - If the result is identical to T, stop; T is a fixed point
  - Otherwise, let T be the new result; repeat

☞ Starting from $\emptyset$ produces the unique minimal fixed point (assuming q is monotone)

Finding ancestors

- WITH RECURSIVE
  Ancestor(anc, desc) AS
  ((SELECT parent, child FROM Parent)
  UNION
  (SELECT a1.anc, a2.desc
  FROM Ancestor a1, Ancestor a2
  WHERE a1.desc = a2.anc))
  - Think of the definition as *Ancestor* = *q*(*Ancestor*)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe

anc	*desc*

→

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe

→

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer

→

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer
Ape	Bart
Ape	Lisa

Intuition behind fixed-point iteration

- Initially, we know nothing about ancestor-descendent relationships
- In the first step, we deduce that parents and children form ancestor-descendent relationships
- In each subsequent steps, we use the facts deduced in previous steps to get more ancestor-descendent relationships
- We stop when no new facts can be proven

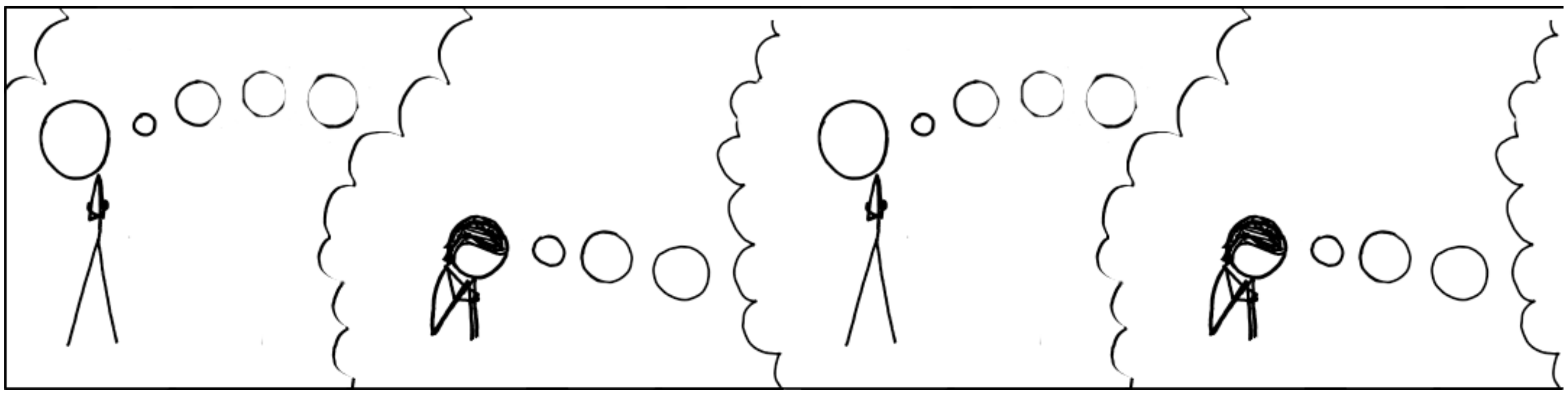
Linear recursion

- With linear recursion, a recursive definition can make only one reference to itself
- Non-linear
  - ```
    WITH RECURSIVE Ancestor(anc, desc) AS
    ((SELECT parent, child FROM Parent)
     UNION
     (SELECT a1.anc, a2.desc
      FROM Ancestor a1, Ancestor a2
      WHERE a1.desc = a2.anc))
    ```
- Linear
  - ```
    WITH RECURSIVE Ancestor(anc, desc) AS
    ((SELECT parent, child FROM Parent)
     UNION
     (SELECT anc, child
      FROM Ancestor, Parent
      WHERE desc = parent))
    ```

Linear vs. non-linear recursion

- Linear recursion is easier to implement
  - For linear recursion, just keep joining newly generated *Ancestor* rows with *Parent*
  - For non-linear recursion, need to join newly generated *Ancestor* rows with all existing *Ancestor* rows

- Non-linear recursion may take fewer steps to converge, but perform more work
  - Example: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$
  - Linear recursion takes 4 steps
  - Non-linear recursion takes 3 steps
    - More work: e.g., $a \rightarrow d$ has two different derivations



http://xkcdsw.com/3080

Mutual recursion example

- Table *Natural* (*n*) contains 1, 2, …, 100
- Which numbers are even/odd?
  - An odd number plus 1 is an even number
  - An even number plus 1 is an odd number
  - 1 is an odd number

```
WITH RECURSIVE Even(n) AS
        (SELECT n FROM Natural
         WHERE n = ANY(SELECT n+1 FROM Odd)),
        RECURSIVE Odd(n) AS
        ((SELECT n FROM Natural WHERE n = 1)
         UNION
         (SELECT n FROM Natural
          WHERE n = ANY(SELECT n+1 FROM Even)))
```

Semantics of WITH

- WITH RECURSIVE R_1 AS Q_1, …,
  RECURSIVE R_n AS Q_n
  Q;
  - Q and $Q_1, \ldots, Q_n$ may refer to $R_1, \ldots, R_n$

- Semantics

  1. $R_1 \leftarrow \emptyset, \ldots, R_n \leftarrow \emptyset$

  2. Evaluate $Q_1, \ldots, Q_n$ using the current contents of $R_1, \ldots, R_n$:
     $R_1^{new} \leftarrow Q_1, \ldots, R_n^{new} \leftarrow Q_n$

  3. If $R_i^{new} \neq R_i$ for some i
     3.1. $R_1 \leftarrow R_1^{new}, \ldots, R_n \leftarrow R_n^{new}$
     3.2. Go to 2.

  4. Compute Q using the current contents of $R_1, \ldots R_n$
     and output the result

Computing mutual recursion

```
WITH RECURSIVE Even(n) AS
        (SELECT n FROM Natural
         WHERE n = ANY(SELECT n+1 FROM Odd)),
        RECURSIVE Odd(n) AS
        ((SELECT n FROM Natural WHERE n = 1)
         UNION
         (SELECT n FROM Natural
          WHERE n = ANY(SELECT n+1 FROM Even)))
```

- *Even* = ∅, *Odd* = ∅
- *Even* = ∅, *Odd* = {1}
- *Even* = {2}, *Odd* = {1}
- *Even* = {2}, *Odd* = {1, 3}
- *Even* = {2, 4}, *Odd* = {1, 3}
- *Even* = {2, 4}, *Odd* = {1, 3, 5}
- …

Fixed points are not unique

```
WITH RECURSIVE
Ancestor(anc, desc) AS
((SELECT parent, child FROM Parent)
 UNION
 (SELECT a1.anc, a2.desc
  FROM Ancestor a1, Ancestor a2
  WHERE a1.desc = a2.anc))
```

parent	child
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Bogus	**Bogus**

anc	desc
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer
Ape	Bart
Ape	Lisa
Bogus	**Bogus**

Note how the bogus tuple reinforces itself!

- But if q is monotone, then all these fixed points must contain the fixed point we computed from fixed-point iteration starting with ∅
- Thus the unique minimal fixed point is the "natural" answer

Mixing negation with recursion

- If q is non-monotone
  - The fixed-point iteration may flip-flop and never converge
  - There could be multiple minimal fixed points—we wouldn't know which one to pick as answer!

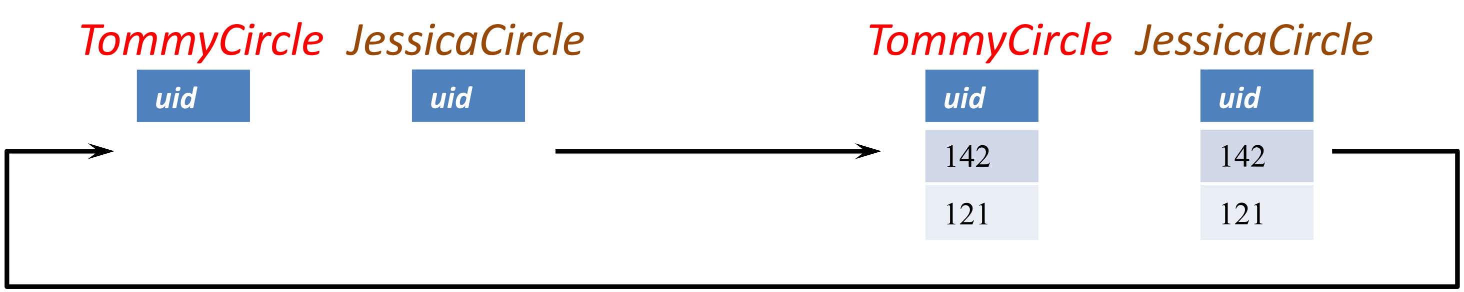
- Example: popular users (pop $\geq$ 0.8) join either Jessica's Circle or Tommy's (but not both)
  - Those not in Jessica's Circle should be in Tom's
  - Those not in Tom's Circle should be in Jessica's

-
```
WITH RECURSIVE TommyCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM JessicaCircle)),
  RECURSIVE JessicaCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
   AND uid NOT IN (SELECT uid FROM TommyCircle))
```

Fixed-point iter may not converge

- WITH RECURSIVE TommyCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
  AND uid NOT IN (SELECT uid FROM JessicaCircle)),
  RECURSIVE JessicaCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
  AND uid NOT IN (SELECT uid FROM TommyCircle))

uid	*name*	*age*	*pop*
142	Bart	10	0.9
121	Allison	8	0.85



Multiple minimal fixed points

- WITH RECURSIVE TommyCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
  AND uid NOT IN (SELECT uid FROM JessicaCircle)),
  RECURSIVE JessicaCircle(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
  AND uid NOT IN (SELECT uid FROM TommyCircle))

uid	*name*	*age*	*pop*
142	Bart	10	0.9
121	Allison	8	0.85

TommyCircle

uid
142

JessicaCircle

uid
121

TommyCircle

uid
121

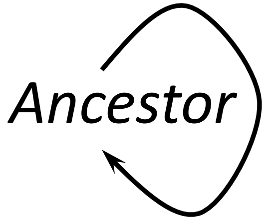
JessicaCircle

uid
142

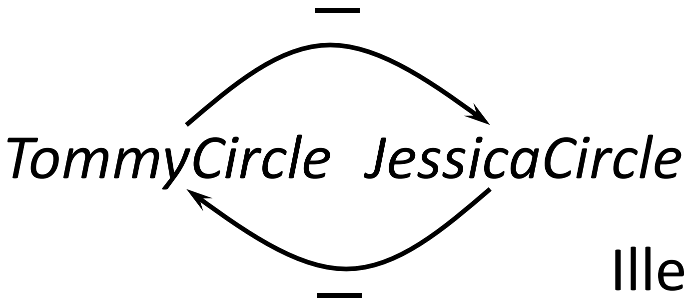
Problem: What do we answer if someone asks whether 121 belongs to JessicaCircle?

Legal mix of negation and recursion

- Construct a dependency graph
  - One node for each table defined in WITH
  - A directed edge $R \rightarrow S$ if R is defined in terms of S
  - Label the directed edge "—" if the query defining R is not monotone with respect to S
- Legal SQL3 recursion: no cycle with a "—" edge
  - Called stratified negation
- Bad mix: a cycle with at least one edge labeled "—"



Legal!

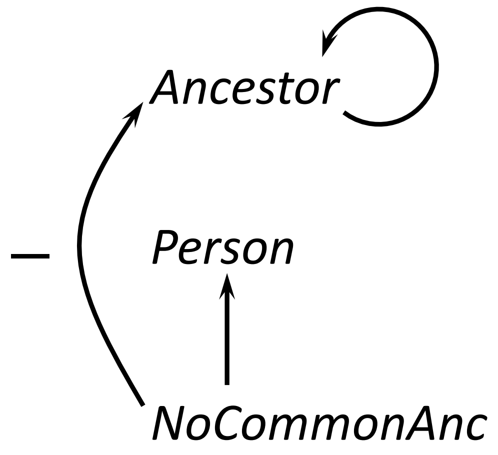


Illegal!

Stratified negation example

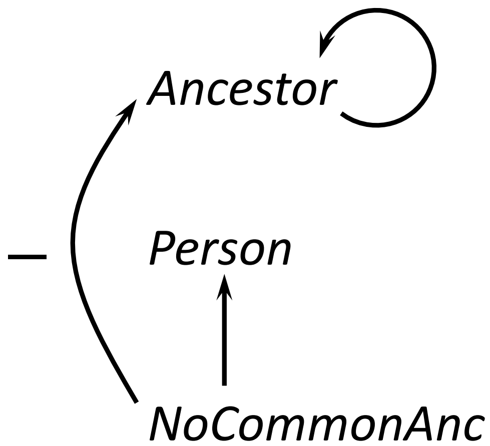
- Find pairs of persons with no common ancestors

```
WITH RECURSIVE Ancestor(anc, desc) AS
   ((SELECT parent, child FROM Parent) UNION
    (SELECT a1.anc, a2.desc
     FROM Ancestor a1, Ancestor a2
     WHERE a1.desc = a2.anc)),
   Person(person) AS
   ((SELECT parent FROM Parent) UNION
    (SELECT child FROM Parent)),
   NoCommonAnc(person1, person2) AS
   ((SELECT p1.person, p2.person
     FROM Person p1, Person p2
     WHERE p1.person <> p2.person)
    EXCEPT
    (SELECT a1.desc, a2.desc
     FROM Ancestor a1, Ancestor a2
     WHERE a1.anc = a2.anc))
SELECT * FROM NoCommonAnc;
```



Evaluating stratified negation

- The stratum of a node R is the maximum number of "—" edges on any path from R in the dependency graph
  - *Ancestor*: stratum 0
  - *Person*: stratum 0
  - *NoCommonAnc*: stratum 1



- Evaluation strategy
  - Compute tables lowest-stratum first
  - For each stratum, use fixed-point iteration on all nodes in that stratum
    - Stratum 0: *Ancestor* and *Person*
    - Stratum 1: *NoCommonAnc*

☞ Intuitively, there is no negation within each stratum

Summary so far

- SQL3 WITH recursive queries
- Solution to a recursive query (with no negation): unique minimal fixed point
- Computing unique minimal fixed point: fixed-point iteration starting from ∅
- Mixing negation and recursion is tricky
  - Illegal mix: fixed-point iteration may not converge; there may be multiple minimal fixed points
  - Legal mix: stratified negation (compute by fixed-point iteration stratum by stratum)
- Another language for recursion: Datalog

Datalog

Datalog: Another query language for recursion

- Ancestor(x, y) :- Parent(x, y)
- Ancestor(x, y):- Parent(x, z), Ancestor(z, y)

- Like logic programming
- Multiple rules
- Same "head" = union
- "," = AND

- Same semantics that we discussed so far

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Recall our drinker example in RC (Lecture 4)

Find drinkers that frequent some bar that serves some beer they like.

$Q(x) = \exists y.\ \exists z.\ \text{Frequents}(x, y) \wedge \text{Serves}(y,z) \wedge \text{Likes}(x,z)$

Drinker example is from slides by Profs. Balazinska and Suciu and the [GUW] book

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Write it as a Datalog Rule

Find drinkers that frequent some bar that serves some beer they like.

RC:
$Q(x) = \exists y.\ \exists z.\ Frequents(x, y) \wedge Serves(y,z) \wedge Likes(x,z)$

Datalog:
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Write it as a Datalog Rule

Find drinkers that frequent some bar that serves some beer they like.

RC:
$Q(x) = \exists y.\ \exists z.\ Frequents(x, y) \wedge Serves(y,z) \wedge Likes(x,z)$

Datalog:
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)

- Quick differences:
  - Uses ":-" not =
  - no need for $\exists$ (assumed by default)
  - Use "," on the right hand side (RHS)
  - Anything on RHS the of :- is assumed to be combined with $\wedge$ by default
  - $\forall$, $\Rightarrow$, not allowed – they need to use negation $\neg$
  - Standard "Datalog" does not allow negation
  - Negation allowed in datalog with negation
- How to specify disjunction (OR / $\vee$)?

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example: OR in Datalog

Find drinkers that (a) either frequent some bar that serves some beer they like, (b) or like beer “BestBeer”

RC:
Q(x) = [$\exists$y. $\exists$z. Frequents(x, y) $\wedge$ Serves(y,z) $\wedge$ Likes(x,z)] $\vee$ [Likes(x, “BestBeer”)]

Datalog:
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, “BestBeer”)

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example: OR in Datalog

Find drinkers that (a) either frequent some bar that serves some beer they like, (b) or like beer "BestBeer", (c) or, frequent bars that "Joe" frequents

RC:

$Q(x) = [\exists y.\ \exists z.\ Frequents(x, y) \land Serves(y,z) \land Likes(x,z)] \lor [Likes(x, \text{"BestBeer"})]$
$\lor [\exists w\ Frequents(x, w) \land Frequents(\text{"Joe"}, w)]$

Datalog:

JoeFrequents(w) :- Frequents("Joe", w)
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, "BestBeer")
Q(x) :- Frequents(x, w), JoeFrequents(w)

- To specify "OR", write multiple rules with the same "Head"
- Next: terminology for Datalog

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Datalog Rules

- Each rule is of the form Head :- Body

- Each variable in the head of each rule must appear in the body of the rule

```
JoeFrequents(w) :- Frequents("Joe", w)
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, "BestBeer")
Q(x) :- Frequents(x, w), JoeFrequents(w)
```

Four rules

Head

Body

Atom

Variable

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

EDBs and IDBs

Tuple in an EDB or an IDB: a FACT

either belongs to a given EDB relation, or is derived in an IDB relation

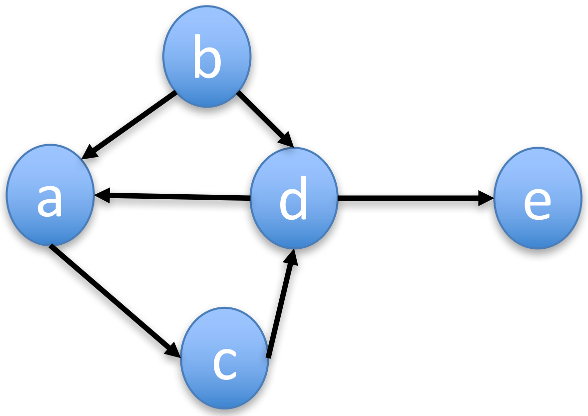
- Extensional DataBases (EDBs)
  - Input relation names
  - e.g. Likes, Frequents, Serves
  - can only be on the RHS of a rule

```
JoeFrequents(w) :- Frequents("Joe", w)
Q(x) :- Frequents(x, y), Serves(y,z), Likes(x,z)
Q(x) :- Likes(x, "BestBeer")
Q(x) :- Frequents(x, w), JoeFrequents(w)
```

- Intensional DataBases (IDBs)
  - Relations that are derived
  - Can be intermediate or final output tables
  - e.g. JoeFrequents, Q
  - Can be on the LHS or RHS (e.g. JoeFrequents)

E (edge relation)

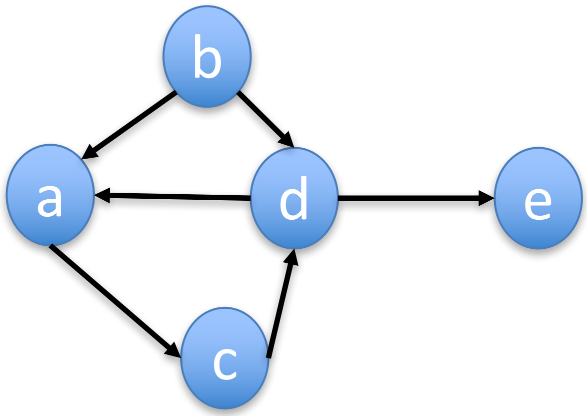
Graph Example



V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

E (edge relation)

Example 1



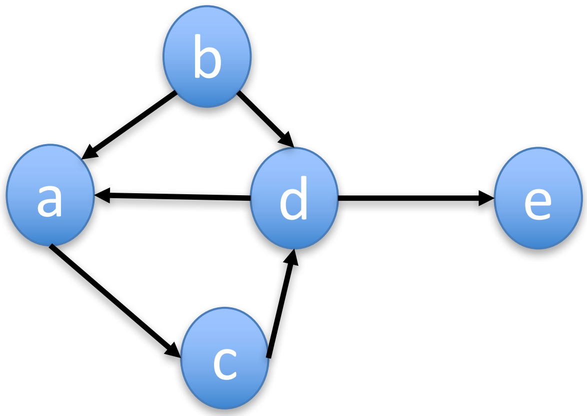
Write a Datalog program to find paths of length two (output start and finish vertices)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

E (edge relation)

Example 1

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e



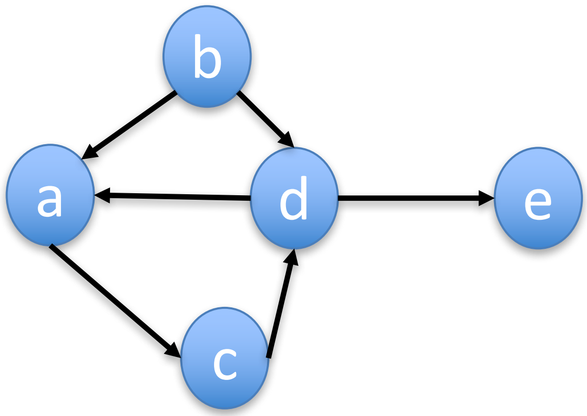
Write a Datalog program to find paths of length two (output start and finish vertices)

P2(x, y) :- E(x, z), E(z, y)

Example 1: Execution

E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e



Write a Datalog program to find paths of length two (output start and finish vertices)

P2(x, y) :- E(x, z), E(z, y)

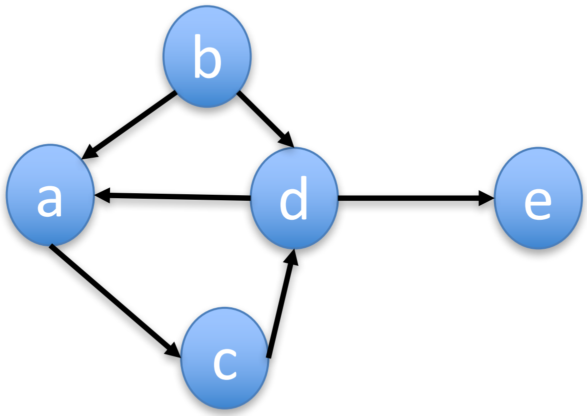
same as $E \bowtie_{E.V2=E.V1} E$

P2

V1	V2
a	d
b	c
b	e
c	a
c	e
d	c

E (edge relation)

Example 2



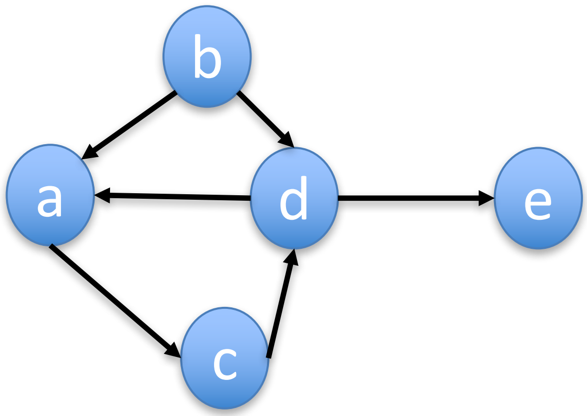
Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

- Can you write a SQL/RA/RC query for reachability?

E (edge relation)

Example 2



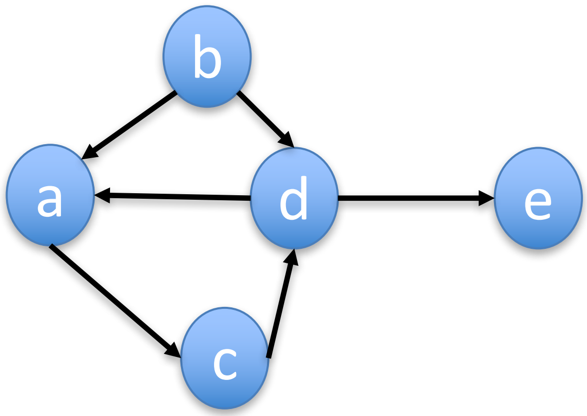
Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

- Not possible in RA/RC

E (edge relation)

Example 2



Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

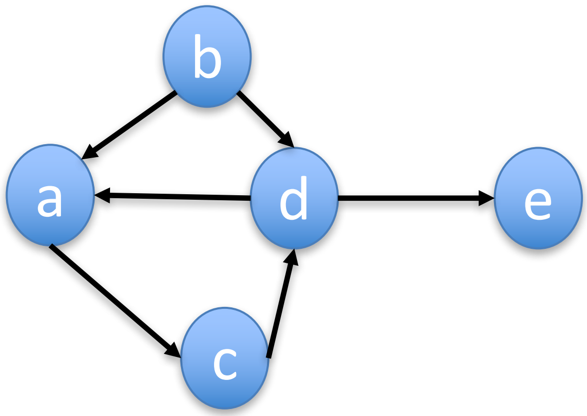
```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
```

Option 1

E (edge relation)

Example 2

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e



Write a Datalog program to find all pairs of vertices (u, v) such that v is reachable from u

Option 1

```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
```

Option 2

```
R(x, y) :- E(x, y)
R(x, y) :- R(x, z), E(z, y)
```

linear (Options 1 and 2)

Option 3

```
R(x, y) :- E(x, y)
R(x, y) :- R(x, z), R(z, y)
```

non-linear (Option 3)

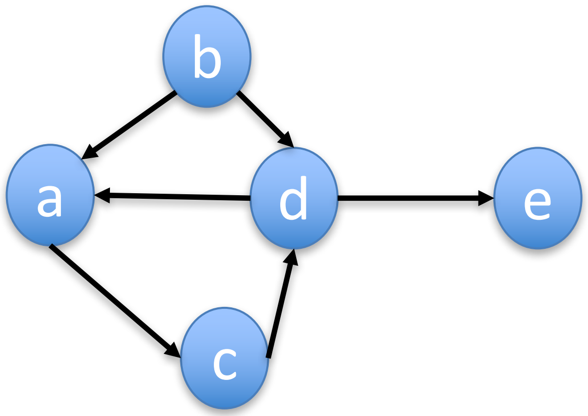
Linear Datalog

- Linear rule
  - at most one atom in the body that is recursive with the head of the rule
  - e.g. R(x, y) :- E(x, z), R(z, y)
- Linear datalog program
  - if all rules are linear
  - like linear recursion

- Top-down and bottom-up evaluation are possible
  - we will focus on bottom-up

Iteration 1

Example 2: Execution



R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

Option 1

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

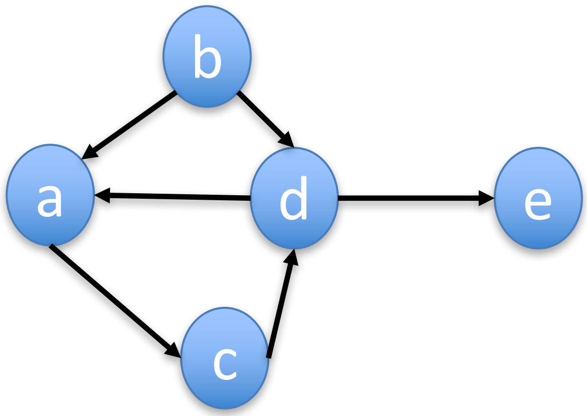
R = E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

(vertices reachable in 1-hop by a direct edge)

Iteration 2

Example 2: Execution



R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

Option 1

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

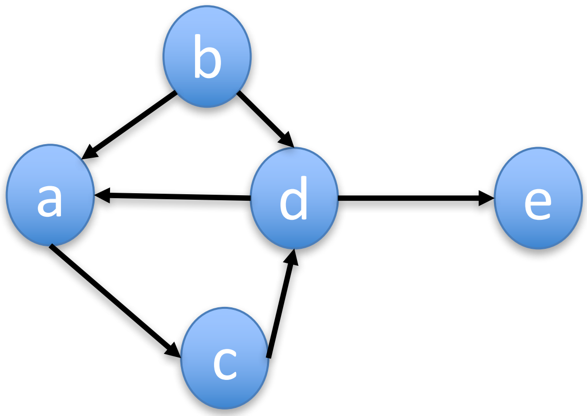
R

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

(vertices reachable in 2-hops)

Iteration 3

Example 2: Execution



R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

Option 1

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

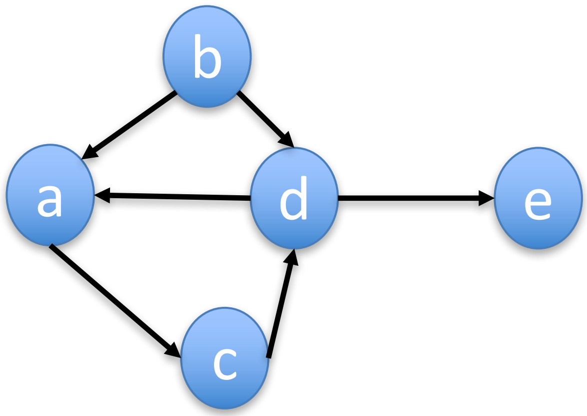
R

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d

(vertices reachable in 3-hops)

Iteration 4

Example 2: Execution



E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)

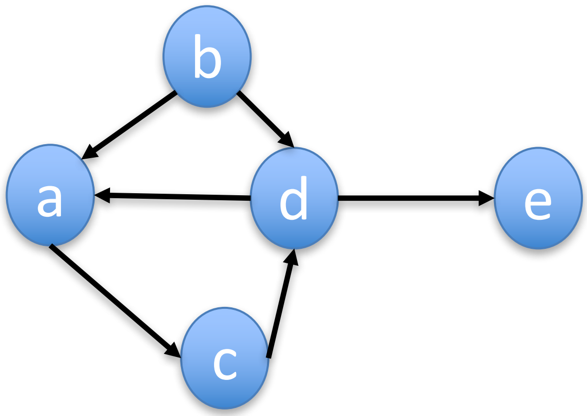
Option 1

R

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d

R unchanged - stop

Examples 3 and 4



E (edge relation)

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Write a Datalog program to find all vertices reachable from b

```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
QB(y) :- R(b, y)
```

Write a Datalog program to find all vertices u reachable from themselves R(u, u)

```
R(x, y) :- E(x, y)
R(x, y) :- E(x, z), R(z, y)
Q(x) :- R(x, x)
```

Termination of a Datalog Program

Q. A Datalog program always terminates – why?

Termination of a Datalog Program

Q. A Datalog program always terminates – why?

- Because the values of the variables are coming from the "active domain" in the input relations (EDBs)

- Active domain = (finite) values from the (possibly infinite) domain appearing in the instance of a database
  - e.g. age can be any integer (infinite), but active domain is only finitely many in R(id, name, age)

- Therefore the number of possible values in each of the IDBs is finite

- e.g. in the reachability example R(x, y), the values of x and y come from {a, b, c, d, e}
  - at most 5 x 5 = 25 tuples possible in the IDB R(x, y)
  - in any iteration, at least one new tuple is added in at least one IDB
  - Must stop after finite steps
  - e.g. the maximum number of iteration in the reachability example for any graph with five vertices is 25 (it was only 4 in our example)

Bottom-up Evaluation of a Datalog Program

- Naïve evaluation
- Semi-naïve evaluation

Naïve evaluation - 1

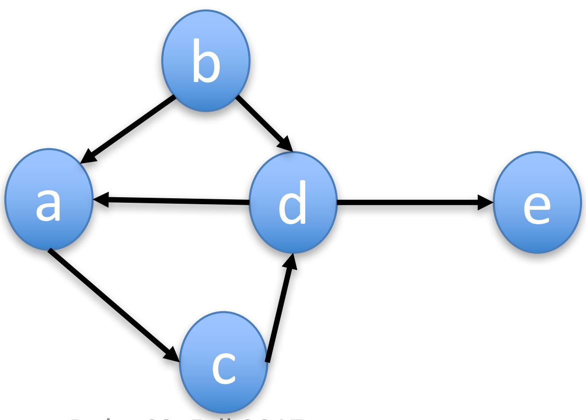
E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:

R = E = R1 (say)



In all subsequent iteration, check if any of the rules can be applied

Do union of all the rules with the same head IDB

Naïve evaluation - 2

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

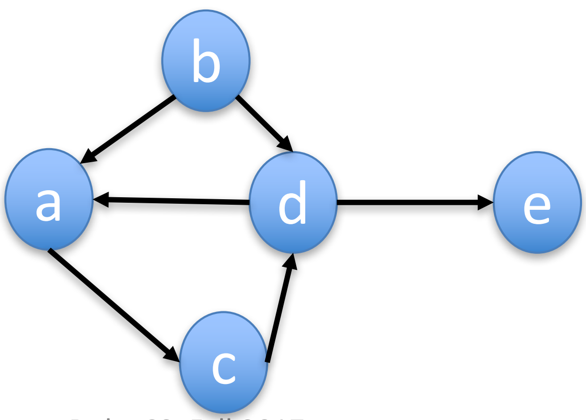
V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)

Iteration 2:
R = **E** ∪ E ⋈ R1 = R2 (say)

R1 ≠ R2
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c



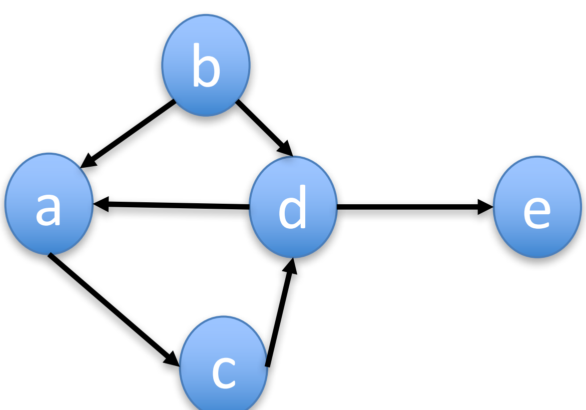
Naïve evaluation - 3

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)



Iteration 2:
R = **E** ∪
E ⋈ R1
= R2 (say)

R1 ≠ R2
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Iteration 3:
R = **E** ∪
E ⋈ R2
= R3 (say)

R2 ≠ R3
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d

Naïve evaluation - 4

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)

Iteration 2:
R = E ∪ E ⋈ R1 = R2 (say)

R1 ≠ R2
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

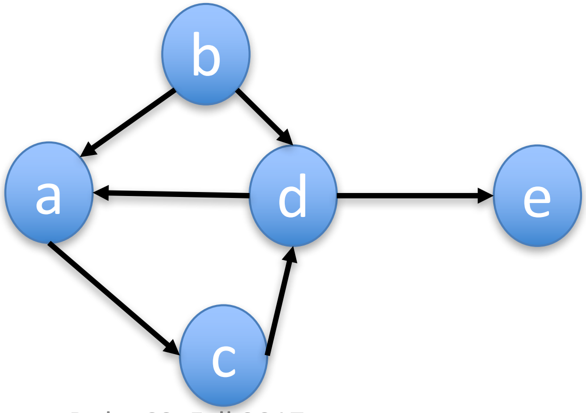
Iteration 3:
R = E ∪ E ⋈ R2 = R3 (say)

R2 ≠ R3
so continue

Iteration 4:
R = E ∪ E ⋈ R3 = R4 (say)

R3 = R4
so STOP

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d



Problem with Naïve Evaluation

- The same IDB facts are discovered again and again
  - e.g. in each iteration all edges in E are included in R
  - In the 2nd-4th iterations, the first six tuples in R are computed repeatedly

- Solution: Semi-Naïve Evaluation

- Work only with the new tuples generated in the previous iteration

Semi-Naïve evaluation - 1

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

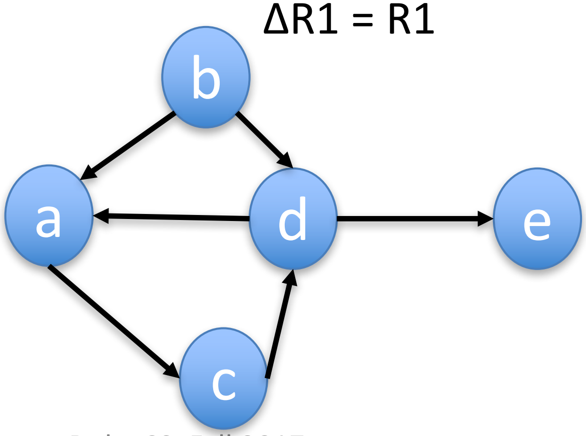
Initially:

R = Φ

Iteration 1:

R = E = R1 (say)

ΔR1 = R1



Semi-Naïve evaluation - 2

E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Initially:

R = Φ

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:

R = E = R1 (say)

ΔR1 = R1

Iteration 2:

R = R1 ∪

E ⋈ ΔR1

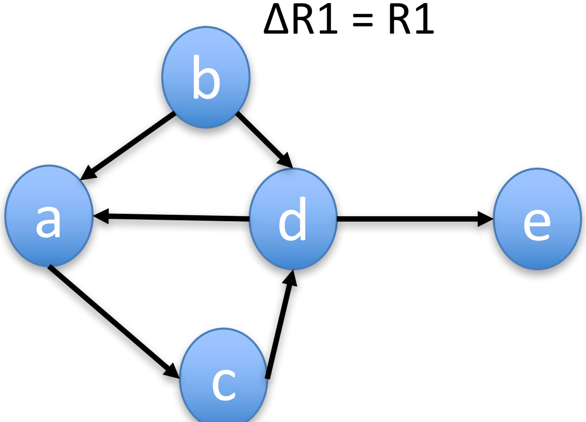
= R2 (say)

ΔR2 = R2 – R1

ΔR2 ≠ Φ

so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c



Semi-Naïve evaluation - 3

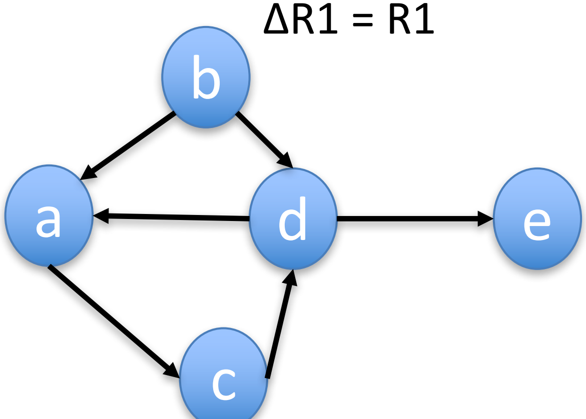
E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Initially:
R = Φ

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)
ΔR1 = R1



Iteration 2:
R = R1 ∪ E ⋈ ΔR1 = R2 (say)

ΔR2 = R2 – R1

ΔR2 ≠ Φ
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Iteration 3:
R = R2 ∪ E ⋈ ΔR2 = R3 (say)

ΔR3 = R3 – R2

ΔR3 ≠ Φ
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d

Semi-Naïve evaluation - 4

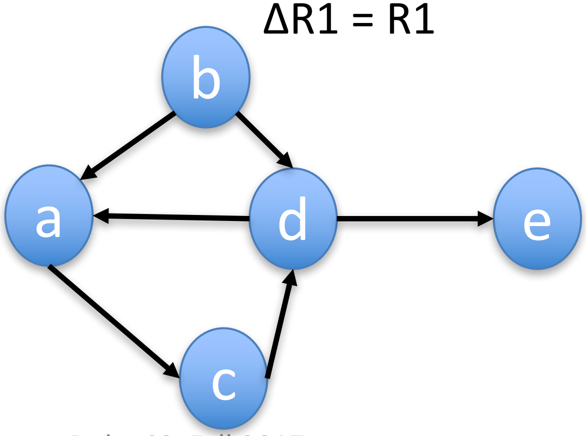
E

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Initially:
R = Φ

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e

Iteration 1:
R = E = R1 (say)
ΔR1 = R1



Iteration 2:
R = R1 ∪
E ⋈ ΔR1
= R2 (say)

ΔR2 = R2 – R1

ΔR2 ≠ Φ
so continue

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c

Iteration 3:
R = R2 ∪
E ⋈ ΔR2
= R3 (say)

ΔR3 = R3 – R2

ΔR3 ≠ Φ
so continue

Iteration 4:
R = R3 ∪
E ⋈ ΔR3
= R4 (say)

ΔR4 = R4 – R3
ΔR = Φ
(CHECK ☺)
so STOP

V1	V2
a	c
b	a
b	d
c	d
d	a
d	e
a	d
b	c
b	e
c	a
c	e
d	c
a	e
a	a
c	c
d	d

Incremental View Maintenance (IVM)

- Why did the semi-naïve algorithm work?
- Because of the generic technique of Incremental "View" Maintenance (IVM)

- What is a view?

Views

- A view is like a "virtual" table
  - Defined by a query, which describes how to compute the view contents on the fly
  - DBMS stores the view definition query instead of view contents
  - Can be used in queries just like a regular table

Creating and dropping views

User(uid, name, pop)
Member(gid, uid)

- Example: members of Jessica's Circle
  - CREATE VIEW JessicaCircle AS
    SELECT * FROM User
    WHERE uid IN (SELECT uid FROM Member
    WHERE gid = 'jes');
  - Tables used in defining a view are called "base tables"
    - *User* and *Member* above
- To drop a view
  - DROP VIEW JessicaCircle;

Using views in queries

- Example: find the average popularity of members in Jessica's Circle
  - SELECT AVG(pop) FROM JessicaCircle;
  - To process the query, replace the reference to the view by its definition
  - SELECT AVG(pop)
    FROM (SELECT * FROM User
    WHERE uid IN
    (SELECT uid FROM Member
    WHERE gid = 'jes'))
    AS JessicaCircle;

```
SELECT AVG(pop)
FROM (SELECT * FROM User
        WHERE uid IN
          (SELECT uid FROM Member
           WHERE gid = 'jes'))
      AS JessicaCircle;
```

Why use views?

- To hide data from users
- To hide complexity from users

- Logical data independence
  - If applications deal with views, we can change the underlying schema without affecting applications

- To provide a uniform interface for different implementations or sources

☞ Real database applications use tons of views

Modifying views

- Does it even make sense, since views are virtual?

- It does make sense if we want users to really see views as tables

- Goal: modify the base tables such that the modification would appear to have been accomplished on the view

A simple case

```
CREATE VIEW UserPop AS
SELECT uid, pop FROM User;
```

```
DELETE FROM UserPop WHERE uid = 123;
```

translates to:

```
DELETE FROM User WHERE uid = 123;
```

An impossible case

```
CREATE VIEW PopularUser AS
   SELECT uid, pop FROM User
   WHERE pop >= 0.8;
```

```
INSERT INTO PopularUser
   VALUES(987, 0.3);
```

- No matter what we do on *User*, the inserted row will not be in *PopularUser*

A case with too many possibilities

```
CREATE VIEW AveragePop(pop) AS
  SELECT AVG(pop) FROM User;
```

– Note that you can rename columns in view definition

```
UPDATE AveragePop SET pop = 0.5;
```

- Set everybody’s *pop* to 0.5?
- Adjust everybody’s *pop* by the same amount?
- Just lower Jessica’s *pop*?

SQL92 updateable views

- More or less just single-table selection queries
  - No join
  - No aggregation
  - No subqueries
  - Other restrictions like "default/ no NOT NULL" values for attributes that are projected out in the view
    - so that they can be extended with valid/NULL values in the base table
- Arguably somewhat restrictive
- Still might get it wrong in some cases
  - See the slide titled "An impossible case"
  - Adding WITH CHECK OPTION to the end of the view definition will make DBMS reject such modifications

End of Lecture 21

INSTEAD OF triggers for views

```
CREATE TRIGGER AdjustAveragePop

INSTEAD OF UPDATE ON AveragePop

REFERENCING OLD ROW AS o,
        NEW ROW AS n

FOR EACH ROW

UPDATE User

SET pop = pop + (n.pop-o.pop);
```

Not covered in detail in this class

Start of Lecture 22

Incremental View Maintenance (IVM)

- Why did the semi-naïve algorithm work?
- Because of the generic technique of Incremental View Maintenance (IVM)

- Suppose you have
  – a database D = (R1, R2, R3)
  – a query Q that gives answer Q(D)
  – D = (R1, R2, R3) gets updated to D' = (R1', R2', R3')
  – e.g. R1' = R1 $\cup$ ΔR1 (insertion), R2' = R2 - ΔR1 (deletion) etc.

Incremental View Maintenance (IVM)

- Why did the semi-naïve algorithm work?
- Because of the generic technique of Incremental View Maintenance (IVM)

- Suppose you have
  - a database D = (R1, R2, R3)
  - a query Q that gives answer Q(D)
  - D = (R1, R2, R3) gets updated to D' = (R1', R2', R3')
  - e.g. R1' = R1 $\cup$ ΔR1 (insertion), R2' = R2 - ΔR1 (deletion) etc.

- IVM: Can you compute Q(D') using Q(D) and ΔR1, ΔR2, ΔR3 without computing it from scratch (i.e. do not rerun the query Q)?

IVM Example: Selection

V1	V2
a	c
b	a
d	a
c	d

R

$\sigma_{V1=b}$ R

V1	V2
b	a

V1	V2
a	c
b	a
d	a
c	d
b	d
d	e

ΔR (last two rows)

R' = R ∪ ΔR

$\sigma_{V1=b}$ R'

V1	V2
b	a
b	d

V1	V2
b	a

$\sigma_{V1=b}$ R

∪

V1	V2
b	d

$\sigma_{V1=b}$ ΔR

- $\sigma_{V1=b} (R \cup \Delta R) = \sigma_{V1=b} R \cup \sigma_{V1=b} \Delta R$
- It suffices to apply the selection condition only on ΔR
  - and include with the original solution

IVM Example: Projection

V1	V2
a	c
b	a
d	a
d	e

R

$\pi_{V1} R$

V1
a
b
d

V1	V2
a	c
b	a
d	a
d	e
b	d
c	d

ΔR (last two rows: b d, c d)

$R' = R \cup \Delta R$

V1
a
b
d
c

$\pi_{V1} R'$

V1
a
b
d

$\pi_{V1} R$

$\cup$

V1
b
c

$\pi_{V1} \Delta R$

- $\pi_{V1} (R \cup \Delta R) = \pi_{V1} R \cup \pi_{V1} \Delta R$
- It suffices to apply the projection condition only on ΔR
  - and include with the original solution

IVM Example: Join

A	B
a1	b1

$\bowtie$

B	C
b1	c1

=

A	B	C
a1	b1	c1

A	B
a1	b1
a2	b2
a3	b1

ΔR (rows a2 b2 and a3 b1)

$\bowtie$

B	C
b1	c1
b2	c2

ΔS (row b2 c2)

$R' = R \cup \Delta R$

$S' = S \cup \Delta S$

=

A	B	C
a1	b1	c1
a3	b1	c1
a2	b2	c2

=

A	B
a1	b1

$\bowtie$

B	C
b1	c1

$\cup$

A	B
a1	b1

$\bowtie$

B	C
b2	c2

$\cup$

A	B
a2	b2
a3	b1

$\bowtie$

B	C
b1	c1

$\cup$

A	B
a2	b2
a3	b1

$\bowtie$

B	C
b2	c2

$(R \cup \Delta R) \bowtie (S \cup \Delta S)$

$= (R \bowtie S) \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)$

IVM for Linear Datalog Rule

A	B
a1	b1

⋈

B	C
b1	c1

=

A	B	C
a1	b1	c1

A	B
a1	b1
a2	b2
a3	b1

ΔR (rows a2 b2, a3 b1)

R' = R ∪ ΔR

⋈

B	C
b1	c1

S' = S

=

A	B	C
a1	b1	c1
a3	b1	c1

(R ∪ ΔR) ⋈ (S ∪ ΔS)

= (R ⋈ S) ∪ (R ⋈ Δ S) ∪ (ΔR ⋈ S) ∪ (Δ R ⋈ Δ S)

- R(x, y) :- E(x, z), R(z, y)
  - i.e. R_{new} = E ⋈ R
- But E is EDB
  - ΔE = Φ
- Therefore,

E ⋈ (R ∪ ΔR) = (E ⋈ R) ∪ (E ⋈ ΔR)

- It suffices to join with the difference ΔR and include in the result in the previous round E ⋈ R
- Advantage of having "linear rule"

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Unsafe/Safe Datalog Rules

Find drinkers who like beer “BestBeer”

Q(x) :- Likes(x, “BestBeer”)

Find drinkers who DO NOT like beer “BestBeer”

Q(x) :- ¬Likes(x, “BestBeer”)

- What is the problem with this rule?
- What should this rule return?
  - names of all drinkers in the world?
  - names of all drinkers in the USA?
  - names of all drinkers in Durham?

Another Problem with Negation in Datalog Rules

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Domain-dependency is bad

Find drinkers who like beer "BestBeer"

Q(x) :- Likes(x, "BestBeer")

Find drinkers who DO NOT like beer "BestBeer"

Q(x) :- ¬Likes(x, "BestBeer")

- What is the problem with this rule?
- Dependent on "domain" of drinkers
  - domain-dependent
  - infinite answers possible too..
    - keep generating "names"
  - Unsafe rule

Another Problem with Negation in Datalog Rules

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Safe Datalog Rules

Find drinkers who like beer "BestBeer"

```
Q(x) :- Likes(x, "BestBeer")
```

Find drinkers who DO NOT like beer "BestBeer"

```
Q(x) :- ¬Likes(x, "BestBeer")
```

- Solution:
- Restrict to "active domain" of drinkers from the input *Likes* (or *Frequents*) relation
  - "domain-independence" – same finite answer always
- Becomes a "safe rule"

```
Q(x) :- Likes(x, y), ¬Likes(x, "BestBeer")
```