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!

A motivating example

```
Parent (parent, child)
```

- 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.
    ```sql
    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?
Recursion in Databases

- What can we do to overcome the limitation?

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:
  - Listo (information extraction)
  - LogicBlox (enterprise decision automation)
  - Semmle (program analysis)
  - BOON/Dedalus (Berkeley)
  - 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

Recursion in SQL

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

```
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';
```

Fixed point of a function

- If \( f: T \rightarrow T \) 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 \( f(0) = 0/2 = 0 \)

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: \( T' = \emptyset \)
- 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)
SELECT anc, desc
FROM Ancestor
```

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

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 \( R_1, R_2, \ldots, R_n \) AS \( Q \)
  - \( Q \) 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 \( R_1, \ldots, R_n \):
     \( R_1^{\text{new}} \leftarrow Q_1, \ldots, R_n^{\text{new}} \leftarrow Q_n \)
  3. If \( R_i^{\text{new}} \neq R_i \) for some \( i \)
     3.1. \( R_i \leftarrow R_i^{\text{new}} \)
     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 = \( 0 \), Odd = \( 0 \)
  - Even = \( 0 \), 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)

\[\text{Ancestor}(\text{parent}, \text{child}) \cup \text{Ancestor}(\text{a1.anc}, \text{a2.desc})\]

- But if \(q\) is monotone, then all these fixed points must contain the fixed point we computed from fixed-point iteration starting with \(\emptyset\)

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 \((\text{pop} \geq 0.8)\) 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

\[\text{WITH RECURSIVE JessicaCircle(uid) AS}
\{(\text{SELECT uid FROM User WHERE pop \geq 0.8})
\text{AND uid NOT IN (SELECT uid FROM JessicaCircle)),}
\text{RECURSIVE JessicaCircle(uid) AS}
\{(\text{SELECT uid FROM User WHERE pop \geq 0.8})
\text{AND uid NOT IN (SELECT uid FROM JessicaCircle))}\]

Multiple minimal fixed points

- WITH RECURSIVE TommyCircle(uid) AS
  \{(\text{SELECT uid FROM User WHERE pop \geq 0.8})
  \text{AND uid NOT IN (SELECT uid FROM JessicaCircle)),}
  \text{RECURSIVE JessicaCircle(uid) AS}
  \{(\text{SELECT uid FROM User WHERE pop \geq 0.8})
  \text{AND uid NOT IN (SELECT uid FROM JessicaCircle))}\]

Legal mix of negation and recursion

- Construct a dependency graph
  - One node for each table defined in WTH
  - A directed edge \(B \rightarrow S\) if \(B\) is defined in terms of \(S\)
  - Label the directed edge “\(\neg\)” if the query defining \(R\) is not monotone with respect to \(S\)

- Legal SQL3 recursion: no cycle with a “\(\neg\)” edge
  - Called stratified negation

- Bad mix: a cycle with at least one edge labeled “\(\neg\)”

Stratified negation example

- Find pairs of persons with no common ancestors

\[\text{WITH RECURSIVE NoCommonAnc(person1, person2) AS}
\{(\text{SELECT person1 FROM Parent UNION}
\text{SELECT person2 FROM Parent)),}
\text{NoCommonAnc(person1, person2) AS}
\{(\text{SELECT P1.person FROM Parent P1 WHERE (SELECT P2.person FROM Parent P2})
\text{EXCEPT}
\text{SELECT P1.person, P2.person FROM Parent P1, Parent P2})},
\text{SELECT * FROM NoCommonAnc;}}\]
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

- 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 $\emptyset$
- 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: 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

Recall our drinker example in RC (Lecture 4)

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

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

Datalog: $Q(x) : \neg \text{Frequents}(x, y) \land \neg \text{Serves}(y, z) \land \neg \text{Likes}(x, z)$

Write it as a Datalog Rule

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

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

Datalog: $Q(x) : \neg \text{Frequents}(x, y) \land \neg \text{Serves}(y, z) \land \neg \text{Likes}(x, z)$
Write it as a Datalog Rule

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

\[ Q(x) = \exists y, z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

- Uses "\(\exists\)" not =
- no need for \(\exists\) (assumed by default)
- Use "\(\land\)" on the right hand side (RHS)
- Anything on RHS the of \(\land\) is assumed to be combined with \(\lor\) by default
- \(\lor\), \(\land\), not allowed – they need to use negation –
- Standard "Datalog" does not allow negation

- How to specify disjunction (OR / \(\lor\))?

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.

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

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

EDBs and IDBs

- Extensional Data Bases (EDBs)
  - Input relation names
  - e.g. Likes, Frequents, Serves
  - can only be on the RHS of a rule
  
  \[ \text{JoeFrequents}(w) \rightarrow \text{Frequents}(\"Joe\", w) \]
  
  \[ Q(x) \rightarrow \text{Frequents}(x, y), \text{Serves}(y, z), \text{Likes}(x, z) \]
  
  \[ Q(x) \rightarrow \text{Likes}(x, \"BestBeer\") \]

- Intensional Data Bases (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)

Graph Example

E (edge relation)

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>
Write a Datalog program to find paths of length two (output start and finish vertices)

\[
P_2(x, y) : - E(x, z), E(z, y)
\]

Example 1: Execution

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

same as \( E \circ E \) or \( E \times E \)

Example 2

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

• Can you write a SQL/RA/RC query for reachability?
• NO - SQL/RA/RC cannot express reachability
Write a Datalog program to find all pairs of vertices \((u, v)\) such that \(v\) is reachable from \(u\):

**Example 1:**

\[
R(x, y) : E(x, y) \\
R(x, y) : E(x, z), R(z, y)
\]

**Option 1:**

\[
R(x, y) : E(x, z), R(z, y)
\]

**Option 2:**

\[
R(x, y) : E(x, z), R(z, y)
\]

**Option 3:**

\[
R(x, y) : E(x, z), R(z, y)
\]

**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**

\[
E
\]

\[
R = E
\]

**Example 2:**

*Vertices reachable in 1-hop by a direct edge*

\[
R(x, y) : E(x, y) \\
R(x, y) : E(x, z), R(z, y)
\]

**Iteration 2**

\[
E
\]

\[
R = E
\]

**Example 2:**

*Vertices reachable in 2-hops*

\[
R(x, y) : E(x, y) \\
R(x, y) : E(x, z), R(z, y)
\]

**Iteration 3**

\[
E
\]

\[
R
\]

**Example 2:**

*Vertices reachable in 3-hops*

\[
R(x, y) : E(x, y) \\
R(x, y) : E(x, z), R(z, y)
\]

**Iteration 4**

\[
R
\]

**Example 2:**

*Vertices reachable in 3-hops*

\[
R(x, y) : E(x, y) \\
R(x, y) : E(x, z), R(z, y)
\]

**R unchanged - stop**
Examples 3 and 4

Write a Datalog program to find all vertices reachable from b

\[
\begin{align*}
R(x, y) & : E(x, y) \\
R(x, y) & : E(x, z), R(z, y) \\
R(x, y) & : R(u, y) \\
Q(x) & : R(b, x)
\end{align*}
\]

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

\[
\begin{align*}
R(x, y) & : E(x, y) \\
R(x, y) & : E(x, z), R(z, y) \\
Q(x) & : R(x, x)
\end{align*}
\]

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

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

Iteration 1:
\[ R = E = R_1 \text{ (say)} \]

Iteration 2:
\[ \begin{align*}
R &= U \\
&= E \cup R_1 \\
&= R_2 \text{ (say)} \\
R_1 &= R_2 \\
\text{so continue}
\end{align*} \]
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 \( 2^{nd} \) iteration, 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-Naive evaluation - 4

E
V1 V2 V3 V4
a c a c
b a b a
d b d d
c d c d
da e d a
d e d e

Initially: R = ∅
ΔR1 = R1
△R1 = R1

Iteration 1:
R = E = R1
Delta R1 = R1
So continue.

Iteration 2:
R = R1 ∪ E ⨝ ΔR1 = R2
ΔR2 = R2
ΔR2 = ∅
So continue.

Iteration 3:
R = R2 ∪ E ⨝ ΔR2 = R3
ΔR3 = R3
ΔR3 = R2
So continue.

Iteration 4:
R = R3 ∪ E ⨝ ΔR3 = R4
ΔR4 = R4
ΔR4 = R3
So STOP.

Incremental View Maintenance (IVM)

Why did the semi-naive 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

- 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;

Why use 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

<table>
<thead>
<tr>
<th>SQL92 updateable views</th>
</tr>
</thead>
<tbody>
<tr>
<td>• More or less just single-table selection queries</td>
</tr>
<tr>
<td>– No join</td>
</tr>
<tr>
<td>– No aggregation</td>
</tr>
<tr>
<td>– No subqueries</td>
</tr>
<tr>
<td>– Other restrictions like &quot;default/ no NOT NULL&quot; values for attributes that are projected out in the view</td>
</tr>
<tr>
<td>• Arguably somewhat restrictive</td>
</tr>
<tr>
<td>• Still might get it wrong in some cases</td>
</tr>
<tr>
<td>– See the slide titled &quot;An impossible case&quot;</td>
</tr>
<tr>
<td>– Adding WITH CHECK OPTION to the end of the view definition will make DBMS reject such modifications</td>
</tr>
</tbody>
</table>

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 AS
SELECT AVG(pop) AS pop FROM User;
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?

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);
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. \( R1' = R1 \cup \Delta R1 \) (insertion), \( R2' = R2 \Delta R1 \) (deletion) etc.

It suffices to apply the selection condition \( \sigma \)
- and include with the original solution

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

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

Example: Join

\[ R \times \Delta S = (a1 \times b1) \cup (a2 \times b2) \]

\[ \Delta R = (a1 \times b1) \cup (a2 \times b2) \]

\[ R \cup \Delta R = (a1 \times b1) \cup (a2 \times b2) \]

\[ (R \cup \Delta R) \times (S \cup \Delta S) = (R \times S) \cup (R \times \Delta S) \cup (\Delta R \times S) \cup (\Delta R \times \Delta S) \]

IVM for Linear Datalog Rule

- \( R(x, y) = E(x, z), R(z, y) \)
- i.e. \( R_{sem} \in \text{EDB} \)
- But \( E \) is EDB
  - \( \Delta E = \emptyset \)

Therefore,
- \( E \vdash (R \cup \Delta R) = (E \vdash R) \cup (E \vdash \Delta R) \)
- It suffices to join with the difference \( \Delta R \) and include in the result in the previous round \( E \vdash R \)
- Advantage of having “linear rule”
Unsafe/Safe Datalog Rules

Find drinkers who like beer “BestBeer”
\[ Q(x) : Likes(x, “BestBeer”) \]

Find drinkers who DO NOT like beer “BestBeer”
\[ Q(x) : \neg Likes(x, “BestBeer”) \]

• 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?

Problem with Negation in Datalog Rules

Find drinkers who like beer “BestBeer”
\[ Q(x) : Likes(x, “BestBeer”) \]

Find drinkers who DO NOT like beer “BestBeer”
\[ Q(x) : \neg Likes(x, “BestBeer”) \]

• What is the problem with this rule?
• Dependent on “domain” of drinkers
  – domain-dependent
  – infinite answers possible too.
    • keep generating “names”

• Solution:
  • Restrict to “active domain” of drinkers from the input
    Likes (or Frequents) relation
    – “domain-independence” – same finite answer always

• Becomes a “safe rule”
\[ Q(x) : Likes(x, y), \neg Likes(x, “BestBeer”) \]