CompSci 516
Data Intensive Computing Systems

Lecture 21 (a)
Datalog

Instructor: Sudeepa Roy
Announcements

• HW4 due tomorrow (Wed, 4/6) at 11:55 pm
• HW5 to be posted tomorrow
  – due on 04/20
  – no extensions
• Jung’s office hour moved today 5:30-6:30
  – in his office
  – for HW4
So far...

- We learnt most of the “must-know” topics in databases (congratulations!)
  - SQL/RA/RC
  - Normalization
  - Storage, Indexing, Hashing
  - Query Execution and Optimization
  - Parallel DBMS/Map-Reduce/Distributed DBMS
  - Transaction CC and Recovery
  - Standard and new database systems in hws (one more coming!)
Remaining five lectures

• Some advanced topics and research directions in databases
  – both traditional and new
  – unfortunately cannot cover all
• Note: every lecture is included in the final exam
  – e.g. there can be short basic conceptual or “brainstorming” questions on these topics
  – Lectures and lecture slides will suffice
Today

• Datalog (part a)
  – for recursion in database queries
  – A quick look at Incremental View Maintenance (IVM)

• NoSQL (part b)
  – And column stores
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
Brief History of Datalog

• Motivated by Prolog – started back in 1970-80’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]

• Now: 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 and startups (academia and industry):
  – Lixto (information extraction), LogicBlox (enterprise decision automation) and Semmle (program analysis)
  – BOOM/Dedalus (Berlekey), Coral, LDL++
Recall our drinker example in RC (Lecture 13)

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

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

Drinker example is from slides by Profs. Balazinska and Suciu and the [GUW] book
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:
\[ Q(x) :- \text{Frequents}(x, y), \text{Serves}(y, z), \text{Likes}(x, z) \]
Write it as a Datalog Rule

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

RC:
Q(x) = ∃y. ∃z. Frequents(x, y) ∧ Serves(y, z) ∧ Likes(x, z)

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

• Quick differences:
  – Use “:-” not =
  – no need for ∃ (assumed by default)
  – Use “,” on the right hand side (RHS)
  – Anything on RHS the of :- is assumed to be combined with ∧ by default
  – ∀, ⇒, not allowed -- need to use ¬ -- Datalog with negation – “Datalog” does not allow negation
  – How to specify disjunction (OR / ∨)?
Example: OR in Datalog

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

RC:
\[ Q(x) = [\exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z)] \lor \text{Likes}(x, \text{“BestBeer”}) \]

Datalog:
\[
\begin{align*}
Q(x) & :- \text{Frequents}(x, y), \text{Serves}(y, z), \text{Likes}(x, z) \\
Q(x) & :- \text{Likes}(x, \text{“BestBeer”})
\end{align*}
\]
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) = [∃y. ∃z. Frequents(x, y) ∧ Serves(y,z) ∧ Likes(x,z)] ∨ [Likes(x, “BestBeer”)]
∨ [∃w Frequents(x, w) ∧ Frequents(“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)
Each rule is of the form `Head :- Body`

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

Four rules:
- \( \text{JoeFrequents}(w) :- \text{Frequents}(\text{"Joe"}, w) \)
- \( \text{Q}(x) :- \text{Frequents}(x, y), \text{Serves}(y, z), \text{Likes}(x, z) \)
- \( \text{Q}(x) :- \text{Likes}(x, \text{"BestBeer"}) \)
- \( \text{Q}(x) :- \text{Frequents}(x, w), \text{JoeFrequents}(w) \)
EDBs and IDBs

• 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)

Tuple in an EDB or an IDB: a FACT

either belongs to a given EDB relation, or is derived in an IDB relation
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>
Example 1

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

\[
\begin{array}{c|c|c|c|c}
\text{V1} & \text{V2} \\
\hline
\text{a} & \text{c} \\
\text{b} & \text{a} \\
\text{b} & \text{d} \\
\text{c} & \text{d} \\
\text{d} & \text{a} \\
\text{d} & \text{e} \\
\end{array}
\]
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

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

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>e</td>
</tr>
<tr>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>

$E$ (edge relation)
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?
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
Example 2

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

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

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

Option 1
Example 2

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:

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

Option 3:

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

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>

Duke CS, Spring 2016

CompSci 516: Data Intensive Computing Systems
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
Example 2: Execution

Option 1

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

(Vertices reachable in 1-hop by a direct edge)
Example 2: Execution

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

Option 1

(iteration 2)

(vertices reachable in 2-hops)
**Iteration 3**

**Example 2: Execution**

Option 1

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

(vertices reachable in 3-hops)

**E**

<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>

**R**

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>V1</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>V1</td>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>V1</td>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>V1</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>V1</td>
<td>c</td>
<td>e</td>
</tr>
<tr>
<td>V1</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>V1</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>V2</td>
<td>a</td>
<td>e</td>
</tr>
<tr>
<td>V2</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>V2</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>
Example 2: Execution

Iteration 4

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

Option 1

R unchanged - stop
Examples 3 and 4

Write a Datalog program to find all vertices reachable from b

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

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

\[
\begin{align*}
R(x, y) & :\quad E(x, y) \\
R(x, y) & :\quad E(x, z), \ R(z, y) \\
Q(x) & :\quad R(x, x)
\end{align*}
\]
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), 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:
\[ R = E = R1 \text{ (say)} \]

Iteration 2:
\[ R = E \cup E \bowtie R1 = R2 \text{ (say)} \]

\[ R1 \neq R2 \text{ so continue} \]

Duke CS, Spring 2016
Naïve evaluation - 3

### Iteration 1:
- $R = E = R1$ (say)

### Iteration 2:
- $R = E \cup E \bowtie R1$
- $R = R2$ (say)
- $R1 \neq R2$
- so continue

### Iteration 3:
- $R = E \cup E \bowtie R2$
- $R = R3$ (say)
- $R2 \neq R3$
- so continue

---

**Table:**

<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>

**Table:**

<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>

**Table:**

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>e</td>
</tr>
<tr>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>e</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>
Naïve evaluation - 4

Iteration 1:
\( R = E = R_1 \) (say)

Iteration 2:
\( R = E \cup E \bowtie R_1 = R_2 \) (say)
R1 \( \neq R_2 \) so continue

Iteration 3:
\( R = E \cup E \bowtie R_2 = R_3 \) (say)
R2 \( \neq R_3 \) so continue

Iteration 4:
\( R = E \cup E \bowtie R_3 = R_4 \) (say)
R3 = R4 so STOP
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\textsuperscript{nd}-4\textsuperscript{th} 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

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>e</td>
</tr>
</tbody>
</table>

Initially:
R = Φ

Iteration 1:
R = E = R1 (say)
ΔR1 = R1
Semi-Naïve evaluation - 2

Initially:  
\[ R = \emptyset \]

Iteration 1:
\[ R = E = R1 \text{ (say)} \]
\[ \Delta R1 = R1 \]

Iteration 2:
\[ R = R1 \cup E \bowtie \Delta R1 \]
\[ = R2 \text{ (say)} \]
\[ \Delta R2 = R2 - R1 \]
\[ \Delta R2 \neq \emptyset \]
so continue

Duke CS, Spring 2016  
CompSci 516: Data Intensive Computing Systems
Semi-Naïve evaluation - 3

Iteration 1:
- \( R = \emptyset \)
- \( E = R_1 \) (say)
- \( \Delta R_1 = R_1 \)

Iteration 2:
- \( R = R_1 \cup E \Join \Delta R_1 \) = \( R_2 \) (say)
- \( \Delta R_2 = R_2 - R_1 \)
- \( \Delta R_2 \neq \emptyset \) so continue

Iteration 3:
- \( R = R_2 \cup E \Join \Delta R_2 \) = \( R_3 \) (say)
- \( \Delta R_3 = R_3 - R_2 \)
- \( \Delta R_3 \neq \emptyset \) so continue

Initially:
- \( R = \emptyset \)
Semi-Naïve evaluation - 4

Initially:
\[ R = \emptyset \]

Iteration 1:
\[ R = E = R1 \ (say) \]
\[ \Delta R1 = R1 \]

Iteration 2:
\[ R = R1 \cup E \bowtie \Delta R1 = R2 \ (say) \]
\[ \Delta R2 = R2 - R1 \]
\[ \Delta R2 \neq \emptyset \]
so continue

Iteration 3:
\[ R = R2 \cup E \bowtie \Delta R2 = R3 \ (say) \]
\[ \Delta R3 = R3 - R2 \]
\[ \Delta R3 \neq \emptyset \]
so continue

Iteration 4:
\[ R = R3 \cup E \bowtie \Delta R3 = R4 \ (say) \]
\[ \Delta R4 = R4 - R3 \]
\[ \Delta R = \emptyset \] (CHECK 😊)
so STOP

V1 | V2  
---|---
| a | c  
| b | a  
| b | d  
| c | d  
| d | a  
| d | e  

Duke CS, Spring 2016
CompSci 516: Data Intensive Computing Systems
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.
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.

• IVM: Can you compute $Q(D’)$ using $Q(D)$ and $\Delta R1, \Delta R2, \Delta R3$ without computing it from scratch (i.e. do not rerun the query $Q$)?
### IVM Example: Selection

<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>d</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

$R$

<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>d</td>
<td>a</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

$\sigma_{V_1=b} R$

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

$\Delta R$

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

$\sigma_{V_1=b} R' = R \cup \Delta R$

- $\sigma_{V_1=b}(R \cup \Delta R) = \sigma_{V_1=b} R \cup \sigma_{V_1=b} \Delta R$
- It suffices to apply the selection condition only on $\Delta R$ — and include with the original solution
IVM Example: Projection

\[
\pi_{V_1}(R \cup \Delta R) = \pi_{V_1} R \cup \pi_{V_1} \Delta R
\]

- It suffices to apply the projection condition only on \(\Delta R\)
  - and include with the original solution
**IVM Example: Join**

\[
(R \cup \Delta R) \bowtie (S \cup \Delta S)
\]

\[
= (R \bowtie S) \cup (R \bowtie \Delta S) \cup (\Delta R \bowtie S) \cup (\Delta R \bowtie \Delta S)
\]
IVM for Linear Datalog Rule

\[
\begin{array}{c|c|c}
A & B & \times \\
\hline
a1 & b1 & \\
\hline
a2 & b2 & \\
\hline
a3 & b1 & \\
\end{array}
\quad \times \quad 
\begin{array}{c|c|c}
B & C & \\
\hline
b1 & c1 & \\
\end{array}
= 
\begin{array}{c|c|c|c}
A & B & C & \\
\hline
a1 & b1 & c1 & \\
\end{array}
\]

**R’ = R ∪ ΔR**

\[
\begin{array}{c|c|c|c}
A & B & C & \\
\hline
a1 & b1 & c1 & \\
\hline
a3 & b1 & c1 & \\
\end{array}
\]

- \( R(x, y) :\ E(x, z),\ R(z, y) \)
  - i.e. \( R_{\text{new}} = E \bowtie R \)
- But \( E \) is EDB
  - \( \Delta E = \Phi \)
- Therefore,
  \[ E \bowtie (R ∪ ΔR) = (E \bowtie R) ∪ (E \bowtie ΔR) \]
- It suffices to join with the difference \( ΔR \) and include in the result in the previous round \( E \bowtie R \)
- Advantage of having “linear rule”

\[
(R ∪ ΔR) \bowtie (S ∪ ΔS) = (R \bowtie S) ∪ (R \bowtie ΔS) ∪ (ΔR \bowtie S) ∪ (ΔR \bowtie ΔS)
\]
(Non-recursive) Datalog with Negation

• Recursion and negation together make Datalog execution complicated
  – there is a notion called “stratified semantic” for this purpose – compute IDB relations in strata/layers before taking a negation – not covered in this class

• We will only do negation for non-recursive Datalog
Unsafe/Safe Datalog Rules

Find drinkers who like beer “BestBeer”

$$Q(x) :\text{- Likes}(x, \text{“BestBeer”})$$

Find drinkers who DO NOT like beer “BestBeer”

$$Q(x) :\text{-} \text{Likes}(x, \text{“BestBeer”})$$

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

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”
Problem with Negation in Datalog Rules

Find drinkers who like beer “BestBeer”

\[ Q(x) :\neg \text{Likes}(x, \text{“BestBeer”}) \]

Find drinkers who DO NOT like beer “BestBeer”

\[ Q(x) : \text{Likes}(x, y), \neg \text{Likes}(x, \text{“BestBeer”}) \]

• Solution:
  • Restrict to “active domain” of drinkers from the input Likes (or Frequents) relation
    – “domain-independence” – same finite answer always
  • Becomes a “safe rule”
Query: Find drinkers that like some beer (so much) that they frequent all bars that serve it

\[ Q(x) = \exists y. \text{Likes}(x, y) \land \forall z. (\text{Serves}(z, y) \Rightarrow \text{Frequents}(x, z)) \]
Query: Find drinkers that like some beer so much that they frequent all bars that serve it

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

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

Step 1: Replace \( \forall \) with \( \exists \) using de Morgan’s Laws

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

\[ \neg (\neg P \lor Q) \text{ same as } P \land \neg Q \]

\[ \forall x \ P(x) \text{ same as } \neg \exists x \ \neg P(x) \]

\[ P \implies Q \text{ same as } \neg P \lor Q \]
Query: Find drinkers that like some beer so much that they frequent all bars that serve it

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

Step 1: Replace \( \forall \) with \( \exists \) using de Morgan’s Laws

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

(new) Step 2: Make all subqueries domain independent

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

Ack: slides by Profs. Balazinska and Suciu
RC → Datalog with negation → SQL (4/8)

Q(x) = ∃y. Likes(x, y) ∧ ¬ ∃z. (Likes(x, y) ∧ Serves(z, y) ∧ ¬Frequents(x, z))

H(x, y)

(new) Step 3: Create a datalog rule for some subexpressions of the form
∃x ∃y…. R(....)∧ S(....)∧ T(....)∧....

H(x, y) :- Likes(x, y), Serves(z, y), not Frequents(x, z)
Q(x) :- Likes(x, y), not H(x, y)

Ack: slides by Profs. Balazinska and Suciu
RC → Datalog with negation → SQL (5/8)

Revisit example from Lecture 13

\[
\begin{align*}
H(x,y) & : \neg \text{Likes}(x,y), \text{Serves}(z,y), \neg \text{Frequents}(x,z) \\
Q(x) & : \neg \text{Likes}(x,y), \neg H(x,y)
\end{align*}
\]

Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L WHERE ......
```

Ack: slides by Profs. Balazinska and Suciu
RC → Datalog with negation → SQL (6/8)

H(x,y) :- Likes(x,y), Serves(z,y), not Frequents(x,z)
Q(x) :- Likes(x,y), not H(x,y)

Step 4: Write it in SQL

```sql
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
  (SELECT * FROM Likes L2, Serves S
   WHERE ... ... )
```

Ack: slides by Profs. Balazinska and Suciu
Step 4: Write it in SQL

```
SELECT DISTINCT L.drinker FROM Likes L
WHERE not exists
 (SELECT * FROM Likes L2, Serves S
  WHERE L2.drinker=L.drinker and L2.beer=L.beer
   and L2.beer=S.beer
   and not exists (SELECT * FROM Frequents F
      WHERE F.drinker=L2.drinker
      and F.bar=S.bar))
```

RC → Datalog with negation → SQL (7/8)

H(x,y) :- Likes(x,y), Serves(z,y), not Frequents(x,z)
Q(x) :- Likes(x,y), not H(x,y)

Revisit example from Lecture 13

Ack: slides by Profs. Balazinska and Suciu
Sometimes can simplify the SQL query by using an unsafe datalog rule
Correctness ensured by safe outermost rule

\[
H(x,y) \leftarrow \text{Likes}(x,y), \text{Serves}(z,y), \text{not Frequent}(x,z) \\
Q(x) \leftarrow \text{Likes}(x,y), \text{not } H(x,y)
\]

**Unsafe rule**

**Select**

\[
\text{SELECT DISTINCT } L.\text{drinker} \text{ FROM Likes L} \\
\text{WHERE not exists} \\
(\text{SELECT * FROM Serves S} \\
\text{WHERE L.\text{beer}=S.\text{beer}} \\
\text{and not exists (SELECT * FROM Frequent F} \\
\text{WHERE F.\text{drinker}=L.\text{drinker}} \\
\text{and F.\text{bar}=S.\text{bar})})
\]

Ack: slides by Profs. Balazinska and Suciu
Next - NOSQL – see part (b)