SQL: Recursion

Introduction to Databases
CompSci 316 Fall 2017
Announcements (Tue., Oct. 3)

• **Homework #2** due tonight
  • Deadline extended to Thursday for Problem 6 (Gradiance) only

• **Midterm** in class Thursday
  • Open-book, open-notes
  • Same format as sample midterm
    • Sample solution also posted in Sakai

• **Project Milestone #1** due next Thursday
A motivating example

Parent \((parent, child)\)

<table>
<thead>
<tr>
<th>parent</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homer</td>
<td>Bart</td>
</tr>
<tr>
<td>Homer</td>
<td>Lisa</td>
</tr>
<tr>
<td>Marge</td>
<td>Bart</td>
</tr>
<tr>
<td>Marge</td>
<td>Lisa</td>
</tr>
<tr>
<td>Abe</td>
<td>Homer</td>
</tr>
<tr>
<td>Ape</td>
<td>Abe</td>
</tr>
</tbody>
</table>

• 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

• 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 \( f(0) = 0/2 = 0 \).

• To compute a fixed point of \( f \):
  • 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)
Intuition behind fixed-point iteration

• Initially, we know nothing about ancestor-descendant 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 al.anc, a2.desc
     FROM Ancestor al, Ancestor a2
     WHERE al.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
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 $Q_1$, ..., $Q_n$ may refer to $R_1$, ..., $R_n$

• Semantics

1. $R_1 \leftarrow \emptyset, ..., R_n \leftarrow \emptyset$
2. Evaluate $Q_1$, ..., $Q_n$ using the current contents of $R_1$, ..., $R_n$:
   $R_1^{\text{new}} \leftarrow Q_1$, ..., $R_n^{\text{new}} \leftarrow Q_n$
3. If $R_i^{\text{new}} \neq R_i$ for some $i$
   3.1. $R_1 \leftarrow R_1^{\text{new}}, ..., R_n \leftarrow R_n^{\text{new}}$
   3.2. Go to 2.
4. Compute $Q$ using the current contents of $R_1$, ..., $R_n$ 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 = Ø, Odd = Ø
• Even = Ø, 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 al.anc, a2.desc
FROM Ancestor al, Ancestor a2
WHERE al.desc = a2.anc))

<table>
<thead>
<tr>
<th>parent</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homer</td>
<td>Bart</td>
</tr>
<tr>
<td>Homer</td>
<td>Lisa</td>
</tr>
<tr>
<td>Marge</td>
<td>Bart</td>
</tr>
<tr>
<td>Marge</td>
<td>Lisa</td>
</tr>
<tr>
<td>Abe</td>
<td>Homer</td>
</tr>
<tr>
<td>Ape</td>
<td>Abe</td>
</tr>
<tr>
<td>Abe</td>
<td>Bart</td>
</tr>
<tr>
<td>Abe</td>
<td>Lisa</td>
</tr>
<tr>
<td>Ape</td>
<td>Homer</td>
</tr>
<tr>
<td>Ape</td>
<td>Bart</td>
</tr>
<tr>
<td>Ape</td>
<td>Lisa</td>
</tr>
<tr>
<td>Bogus</td>
<td>Bogus</td>
</tr>
</tbody>
</table>

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 $\emptyset$
  - 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 (pop $\geq 0.8$) join either Jessica’s Circle or Tommy’s
  • 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 $\geq 0.8$
   AND uid NOT IN (SELECT uid FROM JessicaCircle)),

RECURSIVE JessicaCircle(uid) AS
  (SELECT uid FROM User WHERE pop $\geq 0.8$
   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))
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))

<table>
<thead>
<tr>
<th>uid</th>
<th>name</th>
<th>age</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Bart</td>
<td>10</td>
<td>0.9</td>
</tr>
<tr>
<td>121</td>
<td>Allison</td>
<td>8</td>
<td>0.85</td>
</tr>
</tbody>
</table>

TommyCircle JessicaCircle

uid
dir
142
121

TommyCircle JessicaCircle

uid
dir
121
142
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

• 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)