SQL

• Basic SQL (queries, modifications, and constraints)

• Intermediate SQL
  • Triggers
  • Views
  • Indexes

• Advanced SQL
  • Programming
  • Recursion (Optional)
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)
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';
WITH RECURSIVE
Ancestor(anc, desc) AS
(base case)
(SELECT parent, child FROM Parent)
UNION
(recursive step)
(SELECT a1.anc, a2.desc
FROM Ancestor a1, Ancestor a2
WHERE a1.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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>anc</th>
<th>desc</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>Lisa</td>
</tr>
</tbody>
</table>
Fixed point of a function

• If $f: D \rightarrow D$ is a function from a type $D$ to itself, a **fixed point** of $f$ is a value $x$ such that $f(x) = x$
  • Example: what is the fixed point of $f(x) = x/2$?
  • Ans: 0, as $f(0)=0$   With seed 1: 1, 1/2, 1/4, 1/8, 1/16, ... $\rightarrow$ 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
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
Non-linear v.s. linear recursion

• Non-linear

```sql
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: a recursive definition can make only one reference to itself

```sql
WITH RECURSIVE Ancestor2(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: Given \( a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \), i.e., \( a \) is parent of \( b \), \( b \) is parent of \( c \), \ldots, \( d \) is parent of \( e \).
    • The **linear recursion** takes 4 steps to find \((a, e)\) is an ancestor-descendant pair (slide 9, Ancestor2)
    • Question: How about **non-linear recursion**? (slide 9, Ancestor)
Mutual recursion example

• Table Natural \((n)\) contains 1, 2, \ldots, 100
• Which numbers are even/odd?
  • An even number plus 1 is an odd number
  • An odd number plus 1 is an even number
  • 1 is an odd number

```sql
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
   )))
```

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}
• ...
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 $R_1, ..., R_n$:
     $R_1^{new} \leftarrow Q_1, ..., R_n^{new} \leftarrow Q_n$
  3. If $R_i^{new} \neq R_i$ for some $i$
     3.1. $R_1 \leftarrow R_1^{new}, ..., R_n \leftarrow R_n^{new}$
     3.2. Go to 2.
  4. Compute $Q$ using the current contents of $R_1, ..., R_n$
     and output the result
Starting with non-empty set

WITH RECURSIVE
Ancestor(anc, desc) AS
(base case)
(SELECT parent, child FROM Parent)
UNION
(recursive step)
(SELECT a1.anc, a2.desc
FROM Ancestor a1, Ancestor a2
WHERE a1.desc = a2.anc))
.....;
Fixed points are not unique

- If $q$ is monotone, then starting from $\emptyset$ produces the unique minimal fixed point
  - All these fixed points must contain this fixed point
  - $\rightarrow$ the unique minimal fixed point is the “natural” answer

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)
.....;

<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>Abe</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!
Mixing negation with recursion

• If $q$ is non-monotone
  • The fixed-point iteration may never converge
  • There could be multiple minimal fixed points

• Example: popular users ($\text{pop} \geq 0.8$) join either $\text{SGroup}$ or $\text{PGroup}$
  • Those not in $\text{SGroup}$ should be in $\text{PGroup}$
  • Those not in $\text{GGroup}$ should be in $\text{SGroup}$

WITH RECURSIVE $\text{PGroup}$(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
  AND uid NOT IN (SELECT uid FROM $\text{SGroup}$)),
RECURSIVE $\text{SGroup}$(uid) AS
  (SELECT uid FROM User WHERE pop >= 0.8
  AND uid NOT IN (SELECT uid FROM $\text{PGroup}$))
Fixed-point iter may not converge

WITH RECURSIVE PGroup(uid) AS
(SELECT uid FROM User WHERE pop >= 0.8
AND uid NOT IN (SELECT uid FROM SGroup)),
RECURSIVE SGroup(uid) AS
(SELECT uid FROM User WHERE pop >= 0.8
AND uid NOT IN (SELECT uid FROM PGroup))

<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>

PGroup

SGroup

PGroup

SGroup

uid

uid

uid

uid
Multiple minimal fixed points

WITH RECURSIVE PGroup(uid) AS
(SELECT uid FROM User WHERE pop >= 0.8
AND uid NOT IN (SELECT uid FROM SGroup)),
RECURSIVE SGroup(uid) AS
(SELECT uid FROM User WHERE pop >= 0.8
AND uid NOT IN (SELECT uid FROM PGroup))

<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>
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 “−”
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)),
RECURSIVE Person(person) AS
  ((SELECT parent FROM Parent) UNION
   (SELECT child FROM Parent)),
RECURSIVE 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 \( R \)
  - 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

- Basic SQL (queries, modifications, and constraints)
- Intermediate SQL (triggers, views, indexes)
- Programming

- Recursion
  - SQL3 WITH recursive queries
  - Solution to a recursive query (with no negation)
  - Mixing negation and recursion is tricky