CS 348 Lecture 6

Recursion in SQL, Datalog and
SQL Programming

Semih Salihoğlu

Jan 25th, 2022
Outline For Today

1. SQL Recursive Query Support
   - Recursion Motivation & FixedPoint Subroutine
   - WITH and WITH RECURSIVE Clauses
   - Monotonicity
   - Linear vs Non-Linear Recursion
   - Mutual Recursion
   - Important Note About Convergence of Recursive Queries

2. Datalog: A More Elegant Query Languages For Recursion

3. SQL Programming
Strengths and Limitations of SQL So Far

Strengths:

- Excellent fit for tasks using fundamental set operations:
  - projection, joins, filtering, grouping etc. and combinations
- Very high-level:
  1. Declarative: abstracts users away from low-level computations
  2. Physical data independence: abstracts away low-level storage

Limitations:

- Is not Turing-complete
- More specifically: Cannot express recursive computations
- Historically: Recursion was an afterthought when standardizing SQL
Motivating Example 1: Transitive Closure

Ex: Given academic <(co-)supervisor, student> relationships:

Find all academic ancestors/descendants of an academic

<table>
<thead>
<tr>
<th>Advisor</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>supervisor</td>
<td>student</td>
</tr>
<tr>
<td>C1</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Ancestors

<table>
<thead>
<tr>
<th></th>
<th>anc</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Diagram: A → B → C1 → D1 → C2 → D2

Ancestors:

<table>
<thead>
<tr>
<th></th>
<th>anc</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Graph: A, B, C1, C2, D1, D2
Motivating Example 1: Transitive Closure

- Can find ancestors at any fixed degree, e.g., 1st, 2nd or 4th degree
- If max depth d is known: union all possible queries upto degree d:

\[
\text{(SELECT * FROM Advisor) UNION (SELECT Adv1.sup, Adv2.stu FROM Adv1,Adv2 WHERE Adv1.stu=Adv2.sup) UNION ... (SQL Query for d-degree ancestors)}
\]

- But cannot express arbitrary depths
Motivating Example 1: Transitive Closure

- Historical Fact: killer app of graph DBMSs before relational systems was the “parts explosion query” equivalent transitive closure
- Ask me offline if you want to hear more about this history!
Motivating Example 2: Shortest Paths

- Many other queries builds on top of transitive closure.
- Ex: Given flights <from, to, price> relationships:
  - Find cheapest paths from A to F

### Flights

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>2</td>
</tr>
</tbody>
</table>
Motivating Example 2: Shortest Paths

<table>
<thead>
<tr>
<th>Flights</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

- Can find all (shortest) paths with any fixed number, e.g., k, edges
- If max depth \( d \) is known (*and (directed) graph is acyclic*)
  1. Union all paths with up to \( d \) edges. Call this relation AllPaths:
  2. \[\text{SELECT from, to, min(cost) FROM AllPaths}\]
- But cannot express arbitrary depths

3-edge Paths Query:

\[\text{SELECT F1.from, F3.to, F1.cost+F2.cost+F3.cost as cost}\]
\[\text{FROM Flights F1, Flights F2, Flights F3}\]
\[\text{WHERE F1.to=F2.from AND F2.to=F3.from}\]
Transitive closure (TC) and all paths (or shortest paths which depend on all paths) are inherently recursive properties of graphs

Example: TC of v: all nodes that v can directly or indirectly reach

Computing them require a recursive computation subroutine:

High-level Recursive Subroutine for TC:

```
FixedPoint(fnc F w/ T as input):

T_{prev} = ∅
T_{new} = F(T_{prev}) // 1st degree ancestors

while (T_{prev} != T_{new}):
    T_{prev} = T_{new}
    T_{new} = F(T_{prev}) // compute up to next-degree ancestors
```

Equivalently: Compute T_0 = ∅; T_1 = F(T_0); T_2 = F(T_1); ... until T_i = T_{i+1}

Important Questions:

1. When does fp converge?
2. When is it unique?
SQL WITH

A convenient way to define sub-queries and temporary views

WITH

R1 AS Q1
R2 AS Q2
...
Rn AS Qn

Ri is the result of Qi
Ri visible to Ri+1, ..., Rn
Can explicitly specify schema as
R1(foo, bar) AS Q1 o.w inherits from Q

Q // a query that can use existing tables *and* R1, ..., Rn

Ex:

WITH Deg2Anc AS (SELECT Adv1.sup AS anc, Adv2.stu as desc
FROM Advisor Adv1, Advisor Adv2
WHERE Adv1.stu = Adv2.sup)

Deg3Anc AS (....)

SELECT desc FROM (SELECT * FROM Deg2Anc UNION
...
WITH can be suffixed with RECURSIVE keyword

WITH RECURSIVE

\[
\begin{align*}
R_1 & \text{ AS } Q_1 \\
R_2 & \text{ AS } Q_2 \\
\ldots \\
R_n & \text{ AS } Q_n
\end{align*}
\]

Q // a query that can use existing tables *and* R1, ..., Rn

Semantics of “RECURSIVE T AS Q”: run FixedPoint subroutine

\[
\begin{align*}
T_0 &= \emptyset \\
T_1 &= Q \text{ (but use } T_0 \text{ for } T) \\
T_2 &= Q \text{ (but use } T_1 \text{ for } T)
\end{align*}
\]

until } T_i = T_{i+1}

Note: In SQL standard RECURSIVE is bound to specific Ri. We will and some systems bound it to WITH, so all Ri.
WITH RECURSIVE Ancestors(anc, desc) AS (  
  SELECT Ancestor.anc, Adv.stu  
  FROM Ancestor, Advisor  
  WHERE Ancestor.desc = Advisor.sup)  

Problem? Ancestor starts as \(\emptyset\)  

Common fix: UNION with a 2nd query that inits Ancestor to Advisor  

Common WITH RECURSIVE query template:  

WITH RECURSIVE R AS (  
  Q  
  UNION  
  Q  
)
WITH RECURSIVE Ancestors(anc, desc) AS (  
  SELECT sup as anc, stu as desc  
  FROM Advisor  
  UNION  
  SELECT Ancestor.anc, Adv.stu  
  FROM Ancestor, Advisor  
  WHERE Ancestor.desc = Advisor.sup)  

Ancestor_0 = [A, B]  
Q_B = Ancestor_0 \times Advisor  
Q_R = Ancestor_0 \cup Q_B  
Is Ancestor_1 = Ancestor_0 \?  
No: Repeat
TC: ATTEMPT 2: Union w/ a “Base” Case

All 1-degree ancestors

Ancestor_2 =

Q_R = Anc_1 \bowtie Advisor

Is Anc_2 – Ans_1 = \emptyset?
No: Repeat
TC: ATTEMPT 2: Union w/ a “Base” Case

<table>
<thead>
<tr>
<th>Anc₂</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td>anc</td>
<td>desc</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>D1</td>
<td>A</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>D2</td>
<td>B</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All 1- and 2-degree ancestors

Advisor

<table>
<thead>
<tr>
<th>sup</th>
<th>stu</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>C1</td>
<td>D2</td>
</tr>
<tr>
<td>C2</td>
<td>D1</td>
</tr>
<tr>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>B</td>
<td>C1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

Qₐₐₛₚ = Anc₂ ⋈ Advisor

Ancestor₃ =

<table>
<thead>
<tr>
<th>QB</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QR</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>D2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Anc₃</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td>anc</td>
<td>desc</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>D1</td>
<td>A</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>D2</td>
<td>B</td>
<td>D1</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>D2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is Anc₃ – Ans₂ = ∅?
No: Repeat
TC: ATTEMPT 2: Union w/ a “Base” Case

All 1-, 2-, and 3-degree ancestors

\[ \text{Q}_R = \text{Anc}_3 \bowtie \text{Advisor} \]

Ancestor_4 =

\[ \text{Is Anc}_4 - \text{Ans}_3 = \emptyset? \]

Yes: Stop

Final Answer called the fixed point of Q.
Recall common WITH RECURSIVE query template:

```sql
WITH RECURSIVE R AS (Q_B UNION Q_R)
```

Can use other queries/templates (e.g., multiple base cases)

- But some restrictions apply (stay tuned)

Note that fixed-point computation was very well-behaved in TC:

- Computation converged:
  - In finite steps (and computed a finite relation)
  - No oscillations

Question: Are there conditions that guarantee convergence to a unique fixed point of Q?
Monotonicity

- If we focus on core relational algebra foundation of SQL:
  - Select/project/cross product/join/union/set difference/intersection
  - Ignore group by and aggregations and arithmetic functions etc.

- Theorem: If a recursive Q is “monotone w.r.t to every relation it contains”, then Q has a unique and finite fixed point (i.e., the fixed point subroutine is guaranteed to converge)

- Definition: Q is monotone w.r.t R iff adding more tuples to R can not remove tuples from output of Q (but new tuples can appear)
  - i.e., if each t that used to be in the output of Q is guaranteed to remain in output if add more tuples to R (keeping all else same)
Monotonicity

- Recall each core RA operator except set difference is monotone w.r.t their arguments
- E.g.: $R \bowtie_p S$ is monotone w.r.t $R$ and $S$
- But: $R - S$ is non-monotone w.r.t $S$
- Therefore: Any Q that uses core relational algebraic operations and does not use set difference is monotone
  
  => Q will converge to a unique fixed point (if recursive)

- Note: Q can still be monotone even if it contains set difference. But not guaranteed to be.
Why Does Monotonicity Guarantee A Unique Fixed Point For A Recursive Query?

Proof Sketch: Recall fixed point subroutine:

\[ T_0 = \emptyset; \ T_1 = Q \ (\text{but use } T_0 \text{ for } T); \ T_2 = Q \ (\text{but use } T_1 \text{ for } T) \]

\[ \ldots \]

Note we are assuming we are focusing on core RA:

- Each value in a column of \( T_i \) is from a value from base relation
- But a base relation in \( Q \) is finite

Any relation, no matter what its schema is, has a finite maximum size.

B/c \( Q \) is monotone (specifically w.r.t to \( T \)):

\[ T_1 \subset T_2 \subset T_3 \subset \ldots \] (must stop b/c finiteness)

i.e. \( T_1 \subset T_2 \subset \ldots T_k = T_{k+1} \) (and fp stops)

<table>
<thead>
<tr>
<th>Anc_4</th>
<th>Advisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{anc} )</td>
<td>( \text{desc} )</td>
</tr>
<tr>
<td>C1</td>
<td>D1</td>
</tr>
<tr>
<td>C1</td>
<td>D2</td>
</tr>
<tr>
<td>C2</td>
<td>D1</td>
</tr>
<tr>
<td>C2</td>
<td>D2</td>
</tr>
<tr>
<td>B</td>
<td>C1</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>
Example Non-Monotone Recursive Query 1

WITH RECURSIVE \( T(x) \) AS ( 
    SELECT x FROM R 
    UNION 
    SELECT sum(x) as x FROM T 
)

\( R \)

<table>
<thead>
<tr>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

\( T \)

\( T_0 \)

<table>
<thead>
<tr>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

\( T_1 \)

<table>
<thead>
<tr>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

\( T_2 \)

<table>
<thead>
<tr>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

\( T_3 \)

<table>
<thead>
<tr>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

\( T_4 \)

<table>
<thead>
<tr>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

\( \cdots \) would never converge

- Q is non-monotone b/c as we added 3, 3 got deleted, as we added 6, 6 got deleted etc.
- That’s why aggr. not allowed in recursive queries in SQL standard.

A 2\( \text{nd} \) example after we cover “mutual recursion” (stay tuned).
Recall $Q_R$ in transitive closure:

SELECT Ancestor.anc, Adv.stu
FROM Ancestor, Advisor
WHERE Ancestor.desc = Advisor.sup

Has 1 reference to itself Ancestor: Called *linear recursion*

Can have > 1 reference to Ancestor, called *non-linear recursion*
WITH RECURSIVE Ancestors(anc, desc) AS (  
  SELECT sup as anc, stu as desc  
  FROM Advisor  
  UNION  
  SELECT Anc1.anc, Anct.desc  
  FROM Ancestor Anc1, Ancestor Anc2  
  WHERE Ac1.desc = Anc2.anc)
Non-linear Recursive Computation of Ancestors

<table>
<thead>
<tr>
<th>Anc_1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

All 1-degree ancestors

<table>
<thead>
<tr>
<th>Advisor</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sup</td>
<td>stu</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q_B</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q_R</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

∪

<table>
<thead>
<tr>
<th>Anc_2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td>anc</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
### Non-linear Recursive Computation of Ancestors

#### $\text{Anc}_2$

<table>
<thead>
<tr>
<th>anc</th>
<th>desc</th>
<th>anc</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All 1 and 2-degree ancestors

#### $\text{Q}_B$

<table>
<thead>
<tr>
<th>anc</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

#### $\text{Q}_R$

<table>
<thead>
<tr>
<th>anc</th>
<th>desc</th>
<th>anc</th>
<th>desc</th>
<th>anc</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\bigcup$

#### $\text{Anc}_3$

<table>
<thead>
<tr>
<th>a</th>
<th>d</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

=
Non-linear Recursive Computation of Ancestors

All 1, 2, 3, and 4-degree ancestors

<table>
<thead>
<tr>
<th>Anc₃</th>
<th>Advisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>a d</td>
<td>sup</td>
</tr>
<tr>
<td>a d</td>
<td>stu</td>
</tr>
<tr>
<td>a d</td>
<td>1</td>
</tr>
<tr>
<td>a d</td>
<td>2</td>
</tr>
<tr>
<td>a d</td>
<td>3</td>
</tr>
<tr>
<td>a d</td>
<td>4</td>
</tr>
<tr>
<td>a d</td>
<td>5</td>
</tr>
<tr>
<td>a d</td>
<td>6</td>
</tr>
<tr>
<td>a d</td>
<td>7</td>
</tr>
<tr>
<td>a d</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q_B</th>
<th>Q_R</th>
<th>Anc₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>anc</td>
<td>desc</td>
<td>a d</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a d</th>
<th>a d</th>
<th>a d</th>
<th>a d</th>
<th>a d</th>
<th>a d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

\[ \bigcup \]
Non-linear Recursive Computation of Ancestors

\[ \text{Anc}_4 \]

\[
\begin{array}{cccccccc}
\text{a} & \text{d} & \text{a} & \text{d} & \text{a} & \text{d} & \text{a} & \text{d} \\
1 & 2 & 1 & 3 & 2 & 5 & 4 & 8 \\
2 & 3 & 2 & 4 & 3 & 6 & 1 & 6 \\
3 & 4 & 3 & 5 & 4 & 7 & 2 & 7 \\
4 & 5 & 4 & 6 & 5 & 8 & 3 & 8 \\
5 & 6 & 5 & 7 & 1 & 5 & 1 & 7 \\
6 & 7 & 6 & 8 & 2 & 6 & 2 & 8 \\
7 & 8 & 1 & 4 & 3 & 7 & 1 & 8 \\
\end{array}
\]

All 1, … 8-degree ancestors

\[ \text{Q}_B \]

\[
\begin{array}{cc}
\text{anc} & \text{desc} \\
1 & 2 \\
2 & 3 \\
3 & 4 \\
4 & 5 \\
5 & 6 \\
6 & 7 \\
7 & 8 \\
\end{array}
\]

\[ \text{Q}_R \]

\[
\begin{array}{cccccccc}
\text{a} & \text{d} & \text{a} & \text{d} & \text{a} & \text{d} & \text{a} & \text{d} \\
1 & 3 & 2 & 5 & 4 & 8 \\
2 & 4 & 3 & 6 & 1 & 6 \\
3 & 5 & 4 & 7 & 2 & 7 \\
4 & 6 & 5 & 8 & 3 & 8 \\
5 & 7 & 1 & 5 & 1 & 7 \\
6 & 8 & 2 & 6 & 2 & 8 \\
1 & 4 & 3 & 7 & 1 & 8 \\
\end{array}
\]

\[ \text{Anc}_5 \]

\[
\begin{array}{cccccccc}
\text{a} & \text{d} & \text{a} & \text{d} & \text{a} & \text{d} & \text{a} & \text{d} \\
1 & 2 & 1 & 3 & 2 & 5 & 4 & 8 \\
2 & 3 & 2 & 4 & 3 & 6 & 1 & 6 \\
3 & 4 & 3 & 5 & 4 & 7 & 2 & 7 \\
4 & 5 & 4 & 6 & 5 & 8 & 3 & 8 \\
5 & 6 & 5 & 7 & 1 & 5 & 1 & 7 \\
6 & 7 & 6 & 8 & 2 & 6 & 2 & 8 \\
7 & 8 & 1 & 4 & 3 & 7 & 1 & 8 \\
\end{array}
\]

Is \( \text{Anc}_5 - \text{Ans}_4 = \emptyset \)?

Yes: Stop

Fixed Point

Linear recursion would take 8 steps
Linear vs Non-linear Recursion

- For tc-like computations:
  - Linear recursion:
    - Takes *linear* # iterations in the depth of the relationships
    - But each iteration might perform less work b/c joins are between smaller tables
  - Non-linear recursion:
    - Takes logarithmic # iterations in the same depth
    - But each iteration performs more work
  - SQL standard requires/allows linear recursion for performance reasons
Mutual Recursion

- Each Qi in our examples so far referred to itself.
- We can have the following “mutually recursive” set of queries

\[
\text{WITH RECURSIVE}
\]

\[
\text{RECURSIVE R1 AS Q1} \quad \text{e.g. references R2}
\]

\[
\text{RECURSIVE R2 AS Q2} \quad \text{e.g. references R3}
\]

\[
\text{RECURSIVE R3 AS Q3} \quad \text{e.g. references R1}
\]

- No query alone is recursive but Q1, Q2, Q3 together is recursive
- So they need to be executed “in tandem” until fixed point.
Mutual Recursion Example

Table *Natural* (*n*) contains 1

Even/Odd numbers < 100

WITH RECURSIVE

    Even(n) AS (SELECT n FROM Natural
                 WHERE n = ANY(SELECT n+1 FROM Odd) AND n < 100),
    Odd(n) AS (
               (SELECT n FROM Natural WHERE n = 1)
             UNION
               (SELECT n FROM Natural
                 WHERE n = ANY(SELECT n+1 FROM Even) AND n < 100)

Even\(_0\) = \(\emptyset\),\nOdd\(_0\) = \(\emptyset\)
Even\(_1\) = \(\emptyset\),\nOdd\(_1\) = \(\{1\}\)
Even\(_2\) = \(\{2\}\),\nOdd\(_2\) = \(\{1\}\)
Even\(_3\) = \(\{2\}\),\nOdd\(_3\) = \(\{1, 3\}\)
Even\(_4\) = \(\{2, 4\}\),\nOdd\(_4\) = \(\{1, 3\}\)
Even\(_5\) = \(\{2, 4\}\),\nOdd\(_5\) = \(\{1, 3, 5\}\)

...
Example Non-Monotone Recursive Query 2: Set Difference

WITH RECURSIVE PGroup(uid) AS
    (SELECT uid FROM User
     AND uid NOT IN (SELECT uid FROM SGroup)),
RECURSIVE SGroup(uid) AS
    (SELECT uid FROM User
     AND uid NOT IN (SELECT uid FROM PGroup))

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

MINUS can replace with AND uid NOT IN. In general negated sub-queries or MINUS in recursive parts are not allowed.

Q is non-monotone b/c recall set diff. is nonmonotone w.r.t 2nd arg.
Important Note On Monotonicity/Convergence

- In practice: DBMSs will not/cannot check for monotonicity and may allow much more than SQL standard: arithmetic, aggregations.
- Nor will they detect oscillations
- You can write non-converging code. Systems will often run a max # iterations (e.g., 100) and error
- SQL compiler will not error for these errors. This is on the user!
- Be careful with recursive queries: Know your query & database!
Consider this All Paths query Q:

```sql
WITH RECURSIVE AllPaths(s, d, cost) AS
    (SELECT s, d, cost FROM Edges)
    UNION
    (SELECT AllPaths.s, Edges.d, AllPaths.cost+Edges.cost
     FROM AllPaths, Edges
     WHERE AllPaths.d = Edges.s)
```

- If Edges \{ (1, 2, 10) \} => All Paths: \{ (1, 2, 10) \}
- Keep Q the same but add one more tuple (2, 1, 5) to Edges
- Now there are infinitely many (1, 2) and (2, 1) paths:
  \( (1, 2, 10), (1, 2, 25), (1, 2, 40) \) etc..

Systems will allow this query!
Summary of SQL Recursion

- Recursion did not exist from 1986-1999 in SQL Standard
- General Syntax: WITH RECURSIVE
  
  \[R_1 \text{ AS } Q_1\]
  
  \[R_2 \text{ AS } Q_2\]
  
  \[...\]
  
  \[R_n \text{ AS } Q_n\]

- Basic functionality: linear recursion
- Extended functionality: non-linear and mutual recursion
- Unsafe recursive queries: non-monotone (may not converge) queries or query is monotone but output relation’s size is infinite (e.g., due to use of arithmetic)
- Personal opinion: Recursive computations are not elegant in SQL.
A QL based on logical rules of the form: \( \text{Head} := \text{Body} \)

A DB consists of a set of “base relations” (called “extensional” db)

- \( \text{Likes(person, foodItem)} \)
- \( \text{Sells(restaurant, foodItem, cost)} \)
- \( \text{Frequents(person, restaurant)} \)

Ex Rule: \( \text{Happy(p)} := \text{Likes(p, f), Frequents(p, r), Sells(r, f, c)} \)

- Head
- Body: conjunction/AND of “subgoals”

For simplicity: assume head, subgoals can be relation names (called predicates) with arguments that can be variables or constants.

Datalog allows other predicates: e.g., \( c < 20 \)
Semantics of Datalog Rules

- Happy(p) := Likes(p, f), Frequents(p, r), Sells(r, f, c)

- Natural join on common variables:
  - Any p s.t. ”∃ a food f & rest r | p likes f & p frequents r & r sells f”
    is happy
  - In RA: $\Pi_{\text{person}} (\text{Likes} \bowtie \text{Frequents} \bowtie \text{Sells})$

- Equality filters on constants:
  - Happy(p) := Likes(p, f), Frequents(p, r), Sells(r, f, 20)
  - Any p s.t. ”∃ a food f & rest r | p likes f & p frequents r & r sells f
    & x costs 20 CAD” is happy

- Note: also declarative
More “Datalog Program” Examples

- There can be multiple rules with the same head predicate

  Happy(p) := Likes(p, f), Frequents(p, r), Sells(r, f, c)

  Happy(p) := Likes(p, “Chocolate Cake”)

  Happy(p) := Frequents(p, r), Frequents(“Karim”, r)

- Meaning: Any p s.t:

  1) “∃ a food f & rest r | p likes f & p frequents r & r sells f” OR

  2) “p likes Chocolate Cake” OR

  3) “∃ a restaurant r | both Karim and p frequent r”

      is happy!

- Advantage: Arbitrary recursive, non-recursive, mutually recursive rules can be just written down as logical “derivation” rules
More Elegant Recursive Programs

Example 1: Transitive Closure:

Ancestor(a, d) := Advisor(a, d)

Ancestor(a, d) := Ancestor(a, b), Advisor(b, d)

Example 2: Shortest Paths:

AllPaths(a, d, cost) := Edge(a, d, cost)

AllPaths(a, d, totalCost) := AllPaths(a, k, cost1), Edge(k, d, cost2),

\[ \text{totalCost} = \text{cost1} + \text{cost2} \]

ShortestPaths(a, d, min(cost)) := AllPaths(a, k, cost)

- Can be done in SQL WITH RECURSIVE but don’t need to think about any recursive execution.
- Syntax forces one to focus on logical derivation rules for relations.
Given a Datalog program that satisfy some properties (specifically some monotonicity and finiteness rules as before):

\[ R_1 := \text{body 1 (possibly recursive)} \]
\[ R_2 := \text{body 2 (possibly recursive)} \]
\[ \ldots \]
\[ R_2 := \text{body 7 (possibly recursive)} \]
\[ \ldots \]
\[ R_k := \text{body 1000 (possibly recursive)} \]

Apply rules in arbitrary order to generate new tuples and one always converges to same unique fixed-point => i.e., the order of execution does not matter

If you want: run \( R_1 := \text{body 1} \) 500 times if it keeps producing new tuples; then run \( R_2 := \text{body 2} \), then \( \text{Rj} \), then \( R_1 \) again etc.
Several DBMSs, e.g., recent LogicBlox or LinkedIn’s core graph DBMS, adopts Datalog as a query language instead of SQL.

Better fit for apps requiring recursion and logical inference rules (e.g., in knowledge management and traditional AI applications):

\[ \text{Sibling}(x, y) := \text{BioParent}(z, x), \text{BioParent}(z, y), x \neq y \]

Many cool applications have been developed on Datalog: (e.g., declarative distributed network programming)

- See Peter Alvaro’s work from UC Santa Cruz
- Has been the foundation for many seminal theoretical results
Challenge of using SQL on a real app:
- Not intended for general-purpose computation

Solutions
- Augment SQL with constructs from general-purpose programming languages
  - E.g.: SQL/PSM
- Use SQL together with general-purpose programming languages: many possibilities
  - Through an API, e.g., Python psycopg2
  - Embedded SQL, e.g., in C
- SQL generating approaches: Web Programming Frameworks (e.g., Django)
1) Augmenting SQL: SQL/PSM

- An ISO standard to extend SQL to an advanced prog. lang.
  - Control flow, exception handling, etc.
- Several systems adopt SQL/PSM partially (e.g. MySQL, PostgreSQL)
- PSM = Persistent Stored Modules

CREATE PROCEDURE proc_name(param_decls)
   local_decls
   proc_body;

CREATE FUNCTION func_name(param_decls)
   RETURNS return_type
   local_decls
   func_body;

CALL proc_name(params);

Inside procedure body:
SET variable = CALL func_name(params);
CREATE FUNCTION SetMaxPop(IN newMaxPop FLOAT) RETURNS INT
-- Enforce newMaxPop; return # rows modified.
BEGIN
DECLARE rowsUpdated INT DEFAULT 0;
DECLARE thisPop FLOAT;

-- A cursor to range over all users:
DECLARE userCursor CURSOR FOR
  SELECT pop FROM User
  FOR UPDATE;

-- Set a flag upon “not found” exception:
DECLARE noMoreRows INT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND
  SET noMoreRows = 1;

... (see next slide) ...
END

RETURN rowsUpdated;
-- Fetch the first result row:
OPEN userCursor;
FETCH FROM userCursor INTO thisPop;

-- Loop over all result rows:
WHILE noMoreRows <> 1 DO
  IF thisPop > newMaxPop THEN
    -- Enforce newMaxPop:
    UPDATE User SET pop = newMaxPop
    WHERE CURRENT OF userCursor;
  END IF;
  -- Update count:
  SET rowsUpdated = rowsUpdated + 1;
END IF;
-- Fetch the next result row:
FETCH FROM userCursor INTO thisPop;
END WHILE;
CLOSE userCursor;
Other SQL/PSM Features

- Assignment using scalar query results
  - SELECT INTO

- Other loop constructs
  - FOR, REPEAT UNTIL, LOOP

- Flow control
  - GOTO

- Exceptions
  - SIGNAL, RESIGNAL

- ...

- For more PostgreSQL-specific information, look for “PL/pgSQL” in PostgreSQL documentation
2) Working with SQL through an API

- E.g.: Python psycopg2, JDBC, ODBC (C/C++/VB)
- Based on the SQL/CLI (Call-Level Interface) standard
- The application program sends SQL commands to the DBMS at runtime. Gets back a “cursor” that can iterate over results.
- Results are converted to objects in the application program.
  
  Often you use a cursor to loop through result tuples.
import psycopg2
conn = psycopg2.connect(dbname='beers')
cur = conn.cursor()

# list all drinkers:
cur.execute('SELECT * FROM Drinker')
for drinker, address in cur:
    print(drinker + ' lives at ' + address)

# print menu for bars whose name contains "a":
cur.execute('SELECT * FROM Serves WHERE bar LIKE %s', ('%a%',))
for bar, beer, price in cur:
    print('{} serves {} at {:.2f}'.format(bar, beer, price))

cur.close()
conn.close()
# “commit” each change immediately—need to set this option just once at the start of the session
conn.set_session(autocommit=True)

bar = input('Enter the bar to update: ').strip()
beer = input('Enter the beer to update: ').strip()
price = float(input('Enter the new price: '))

try:
    cur.execute('''
    UPDATE Serves
    SET price = %s
    WHERE bar = %s AND beer = %s''', (price, bar, beer))
    if cur.rowcount != 1:
        print('{} row(s) updated: correct bar/beer?\n'.format(cur.rowcount))
except Exception as e:
    print(e)

Perform passing, semantic analysis, optimization, compilation, and finally execution
while true:
# Input bar, beer, price...
cur.execute('''
    UPDATE Serves
    SET price = %s
    WHERE bar = %s AND beer = %s''', (price, bar, beer))

# Check result...

Perform passing, semantic analysis, optimization, compilation, and finally execution

Execute many times
Can we reduce this overhead?
Prepared Statements: Example

cur.execute(''"
    # Prepare once (in SQL)
    PREPARE update_price AS
    # Name the prepared plan,
    UPDATE Serves
    SET price = $1
    # and note the $1, $2, ... notation for
    WHERE bar = $2 AND beer = $3"")
    # parameter placeholders.

while true:
    # Input bar, beer, price...
    cur.execute(''
        EXECUTE update_price(%s, %s, %s),
        (price, bar, beer))....
    # Check result...
Watch Out For SQL Injection Attacks!

The school probably had something like:

```python
cur.execute("SELECT * FROM Students \
WHERE (name = " + name + ")")
```

where `name` is a string input by user

Called an SQL injection attack. Most APIs have ways to sanitize inputs.
Augmenting SQL vs. Programming Through an API

 Pros of augmenting SQL:

- More processing features for DBMS
- More application logic can be pushed closer to data

 Cons of augmenting SQL:

- SQL is already too big
- Complicate optimization and make it impossible to guarantee safety
3) “Embedding” SQL in a host language

- Can be thought of as the opposite of SQL/PSM
- Extends a host language, e.g., C or Java, with SQL-based features
- Can compile host language together with SQL statements and catch SQL errors during *application compilation time*
4) Web Programming Frameworks

- A web development “framework” e.g., Django or Ruby on Rails
- Very frequent approach to web apps that need a DB
- For most parts, no explicitly writing SQL is needed:
- Example: Django Web App Programming:
  - Define “Models”: python objects and only do oo programming
  - Models will be backed up with Relations in an RDBMS
- E.g.: a Person class/object with first and lastName:

```python
from django.db import models

class Person(models.Model):
    f_name = models.CharField(max_len=30)
    l_name = models.CharField(max_len=30)
```

Would lead the “framework” (not the user) to generate the following SQL code somewhere in the web application files:

```sql
CREATE TABLE myapp_person (  
    "id" serial NOT NULL PRIMARY KEY,  
    "f_name" varchar(30) NOT NULL,  
    "l_name" varchar(30) NOT NULL )
```
Thank You