SQL: Recursion (and index in SQL)

Introduction to Databases

CompSci 316 Spring 2017

DUKE
COMPUTER SCIENCE

Announcements (Wed., Feb. 15)

- Homework #2 due Friday 02/17
  - Except Problem 6 (Gradiance) and Problem X1 (non-Gradiance) : due on Thursday 02/23
  - Please submit on time – solutions will be posted by Saturday morning
  - Timeline is tight for the midterm

- Midterm next Wednesday 02/22 in class
  - Up to lecture 9 included
  - Will review some concepts/practice problems on Monday

Indexes

- An index is an auxiliary persistent data structure
  - Search tree (e.g., B⁺-tree), lookup table (e.g., hash table), etc.
  - ☞ More on indexes later in this course!
- An index on $R.A$ can speed up accesses of the form
  - $R.A = value$
  - $R.A > value$ (sometimes; depending on the index type)
- An index on $(R.A_1, \ldots, R.A_n)$ can speed up
  - $R.A_1 = value_1 \wedge \cdots \wedge R.A_n = value_n$
  - $(R.A_1, \ldots, R.A_n) > (value_1, \ldots, value_n)$ (again depends)

☞ Ordering or index columns is important—is an index on $(R.A, R.B)$ equivalent to one on $(R.B, R.A)$?

☞ How about an index on $R.A$ plus another on $R.B$?

Examples of using indexes: 1/2

- SELECT * FROM User WHERE name = 'Bart';

- Without an index on *User.name*:
- must scan the entire table if we store *User* as a flat file of unordered rows

- With an index on *User.name*:
- go "directly" to rows with name='Bart'

Examples of using indexes: 2/2

- SELECT * FROM User, Member
  WHERE User.uid = Member.uid
  AND Member.gid = 'jes';

Recall the semantic for SQL evaluation!

- With an index on *Member.gid* or (*gid*, *uid*):
- find relevant *Member* rows directly

- With an index on *User.uid*:
- for each relevant *Member* row, directly look up *User* rows with matching *uid*

- Without an index:
- for each *Member* row, scan the entire *User* table for matching *uid*
  - Sorting could help

Creating and dropping indexes in SQL

CREATE [UNIQUE] INDEX $indexname$ ON $tablename(columnname_1,\ldots,columnname_n)$;

- With UNIQUE, the DBMS will also enforce that $\{columnname_1, \ldots, columnname_n\}$ is a key of $tablename$

DROP INDEX $indexname$;

- Typically, the DBMS will automatically create indexes for PRIMARY KEY and UNIQUE constraint declarations

Choosing indexes to create

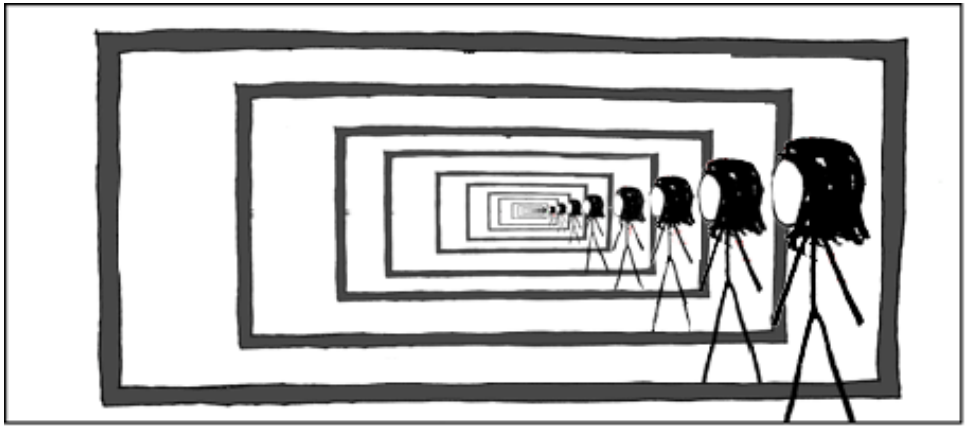
More indexes = better performance?

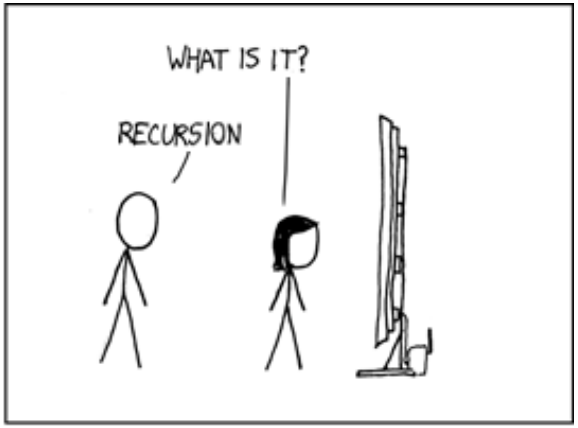
- Indexes take space
- Indexes need to be maintained when data is updated
- Indexes have one more level of indirection

☞ Optimal index selection depends on both query and update workload and the size of tables

- Automatic index selection is now featured in some commercial DBMS

Next: Recursion!



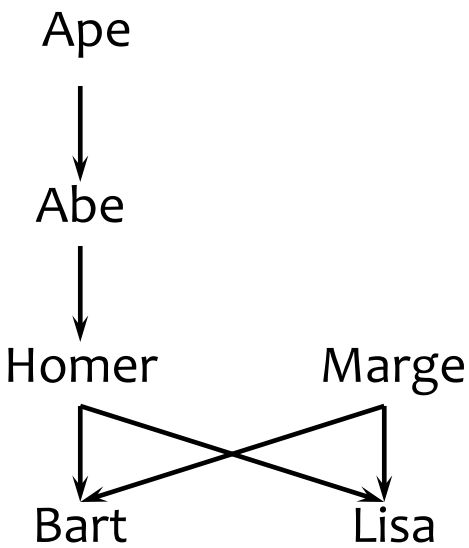


http://xkcdsw.com/1105

A motivating example

Parent (*parent*, *child*)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe



- 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.

```
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

Define a a relation recursively

```
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';
```

base case

recursion step

Query using the relation defined in WITH clause

Fixed point of a function

- If $f: T \to T$ 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 fixed point of a function 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*)

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe

anc	*desc*

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer
Ape	Bart
Ape	Lisa

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

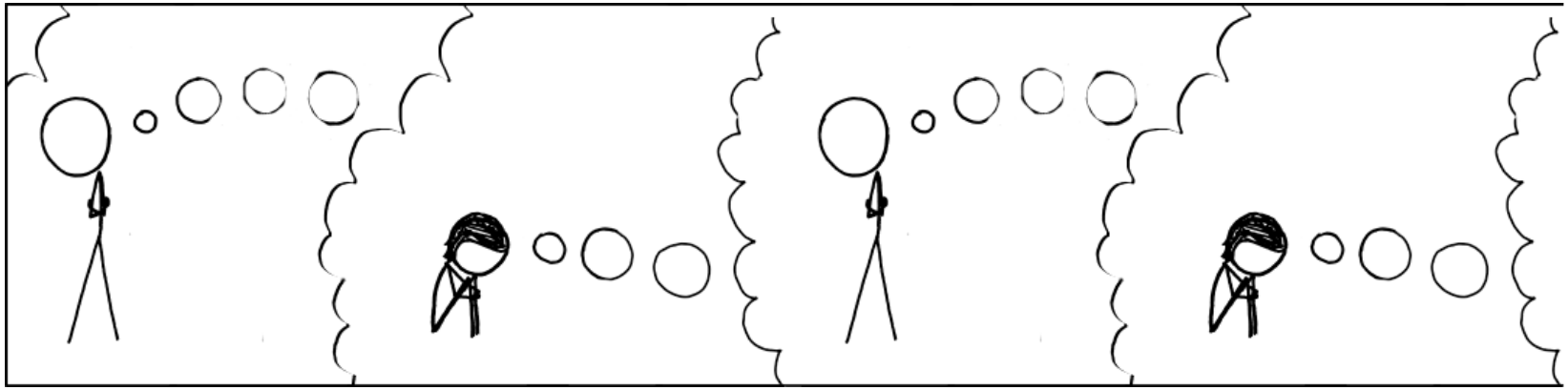
Linear recursion

- 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*
  - 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



http://xkcdsw.com/3080

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 $Q_1, \ldots, Q_n$ 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 $R_1, \ldots, R_n$:
     $R_1^{new} \leftarrow Q_1, \ldots, R_n^{new} \leftarrow Q_n$

  3. If $R_i^{new} \neq R_i$ for some i
     3.1. $R_1 \leftarrow R_1^{new}, \ldots, R_n \leftarrow R_n^{new}$
     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* = ∅, *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}
- …

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))
```

parent	*child*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Bogus	**Bogus**

anc	*desc*
Homer	Bart
Homer	Lisa
Marge	Bart
Marge	Lisa
Abe	Homer
Ape	Abe
Abe	Bart
Abe	Lisa
Ape	Homer
Ape	Bart
Ape	Lisa
Bogus	**Bogus**

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 ≥ 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

- ```
  WITH RECURSIVE TommyCircle(uid) AS
    (SELECT uid FROM User WHERE pop >= 0.8
     AND uid NOT IN (SELECT uid FROM JessicaCircle)),

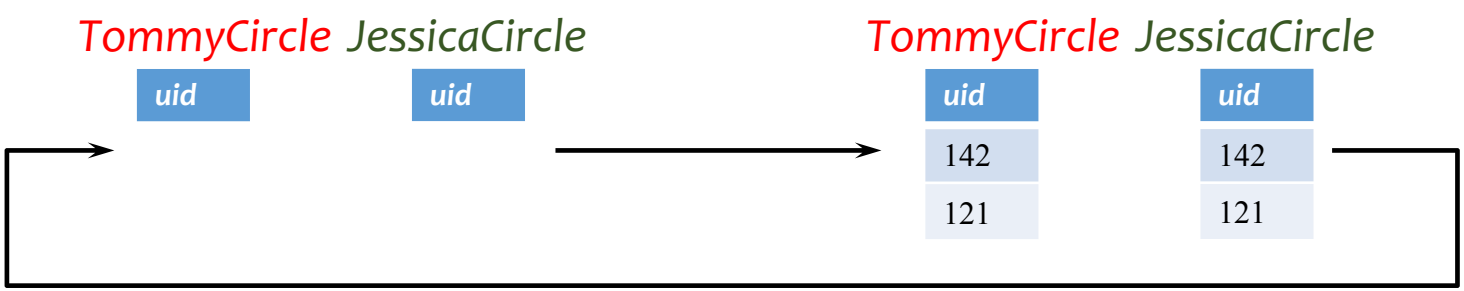
    RECURSIVE JessicaCircle(uid) AS
    (SELECT uid FROM User WHERE pop >= 0.8
     AND uid NOT IN (SELECT uid FROM TommyCircle))
  ```
  ```

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))

uid	name	age	pop
142	Bart	10	0.9
121	Allison	8	0.85

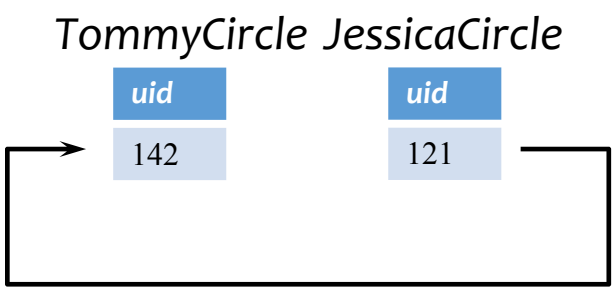


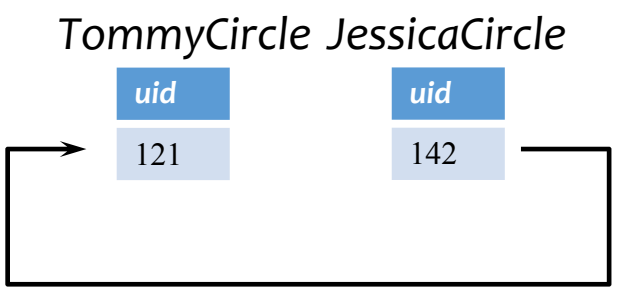
Multiple minimal fixed points

- 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))

uid	name	age	pop
142	Bart	10	0.9
121	Allison	8	0.85





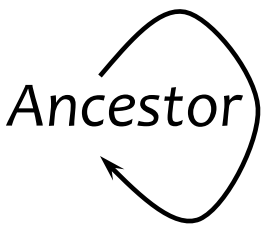
Problem: What do we answer if someone asks whether 121 belongs to JessicaCircle?

Legal mix of negation and recursion

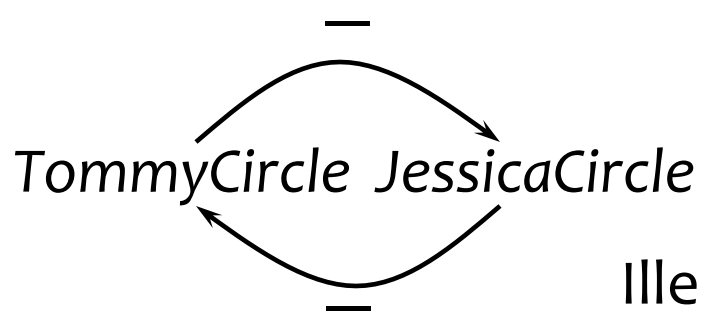
- Construct a dependency graph
  - One node for each table defined in WITH
  - A directed edge $R \rightarrow S$ if R is defined in terms of S
  - Label the directed edge "–" if the query defining R is not monotone with respect to S

- Legal SQL3 recursion: no cycle with a "–" edge
  - Called stratified negation

- Bad mix: a cycle with at least one edge labeled "–"



Legal!



Illegal!

END OF LECTURE 10

Stratified negation example

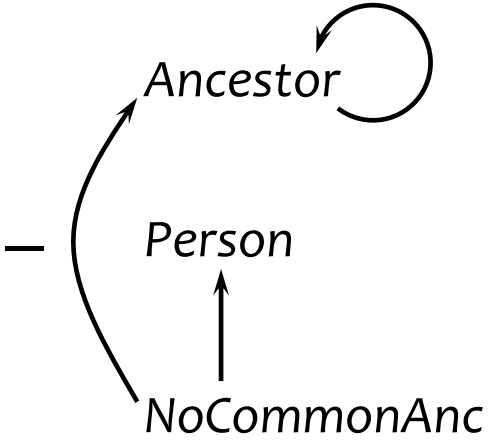
- Find pairs of persons with no common ancestors

```
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)),

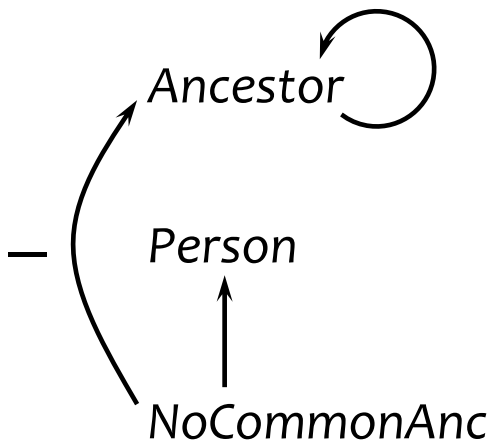
    NoCommonAnc(person1, person2) AS
    ((SELECT p1.person, p2.person
      FROM Person p1, Person p2
      WHERE p1.person <> p2.person)
     EXCEPT
     (SELECT a1.desc, a2.desc
      FROM Ancestor a1, Ancestor a2
      WHERE a1.anc = a2.anc))

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

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
  - not covered in detail in this class

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 Ø
- 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