CS 348 Lecture 8

Recursion in SQL & (optional) Datalog

Semih Salihoğlu

Jan 29th, 2025

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

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:
  - I. Declarative: abstracts users away from low-level computations
  - II. Physical data independence: abstracts away low-level storage

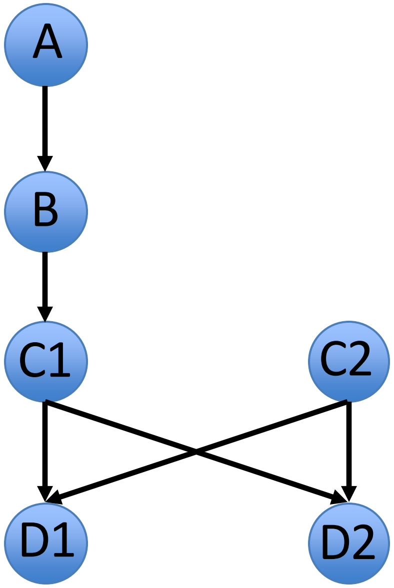
Limitations:

- Is not Turing-complete
- More specifically: Cannot express recursive computations
- Historically: Recursion was an afterthougt when standardizing SQL

Motivating Example 1: Transitive Closure

- Ex: Given academic <(co-)supervisor, student> relationships:
  - Find all academic ancestors/descendants of an academic

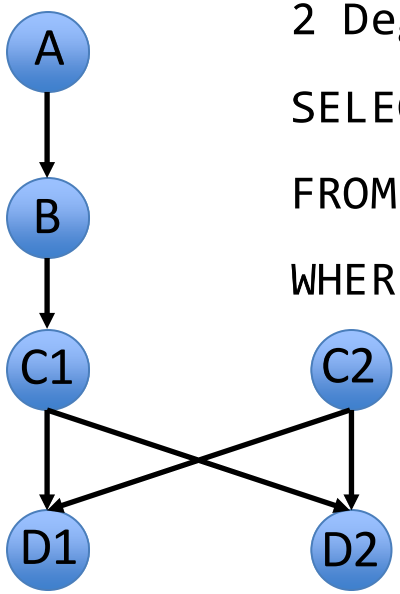
Advisor	
supervisor	student
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B



Ancestors	
anc	desc
…	…
…	…
…	…
…	…
…	…
…	…

Motivating Example 1: Transitive Closure

Advisor	
sup	stu
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B




```
2 Degree Ancestors Query:

SELECT Adv1.sup AS anc, Adv2.stu as desc

FROM Advisor Adv1, Advisor Adv2

WHERE Adv1.stu = Adv2.sup
```

- 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:

```
(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

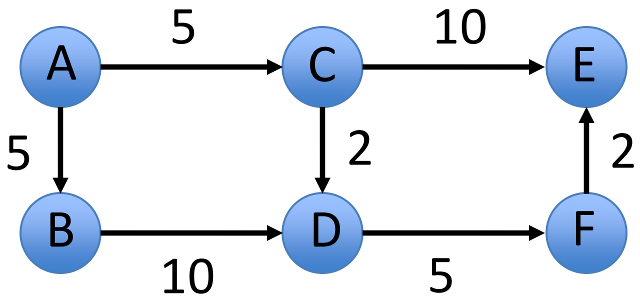
Motivating Example 1: Transitive Closure

- 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

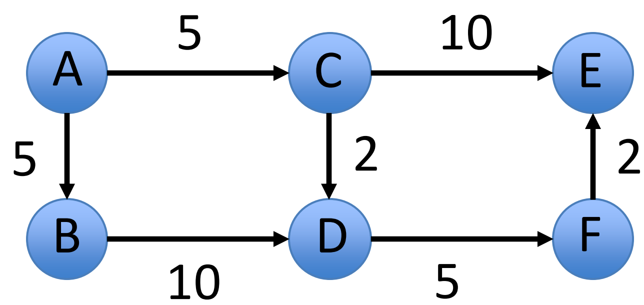
- Many other queries build on top of transitive closure.
- Ex: Given flights <from, to, price> relationships:
  - Find cheapest paths from A to F

Flights		
from	to	cost
A	B	5
A	C	5
B	D	10
C	D	2
C	E	10
D	F	5
F	E	2



Motivating Example 2: Shortest Paths

Flights		
from	to	cost
A	B	5
A	C	5
B	D	10
C	D	2
C	E	10
D	F	5
F	E	2




```
3-edge Paths Query:
SELECT F1.from, F3.to, F1.cost+F2.cost+F3.cost as cost
FROM Flights F1, Flights F2, Flights F3
WHERE F1.to=F2.from AND F2.to=F3.from
```

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

Solution: Recursive "Fixed Point" Computations

- 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 = Ø
    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
```

Important Questions:

1. *When does fp converge?*
2. *When is it unique?*

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

- Upshot: SQL WITH RECURSIVE is a way to run FP subroutine

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
                  SELECT * FROM Deg3Anc)
                 WHERE anc = “A”
```

SQL WITH RECURSIVE

- WITH can be suffixed with RECURSIVE keyword

```
WITH  RECURSIVE
        R1 AS Q1     ——→  Can reference R1
        R2 AS Q2
        …
        Rn AS Qn
```

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

- Semantics of "WITH RECURSIVE T AS Q": run FixedPoint subroutine

$T_0 = \emptyset$

T_1 = Q (but use T_0 for T)

T_2 = Q (but use T_1 for T)

… until $T_i = T_{i+1}$

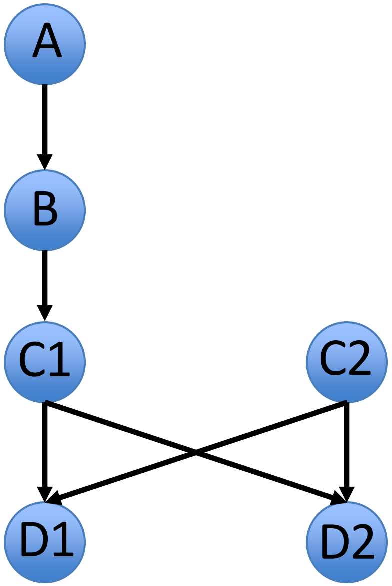
Note: In SQL standard RECURSIVE is bound to specific Ri. We will and some systems bind it to WITH, so all Ri.

TC: ATTEMPT 1

```
WITH RECURSIVE Ancestors(anc, desc) AS (
    SELECT Ancestor.anc, Adv.stu
    FROM Ancestor, Advisor
    WHERE Ancestor.desc = Advisor.sup)
```

Advisor

sup	stu
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B



- Problem? Ancestor starts as ∅
- Common fix: UNION with a 2nd query that inits Ancestor to Advisor
- Common WITH RECURSIVE query template:

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

Q_B: non-recursive "base" query

Q_R: recursive query

TC: ATTEMPT 2: Union w/ a “Base” Case

```
WITH RECURSIVE Ancestors(anc, desc) AS (
    SELECT sup as anc, stu as desc          -- base query
    FROM Advisor
  UNION
    SELECT Ancestor.anc, Adv.stu            -- recursive query
    FROM Ancestor, Advisor
    WHERE Ancestor.desc = Advisor.sup)
```

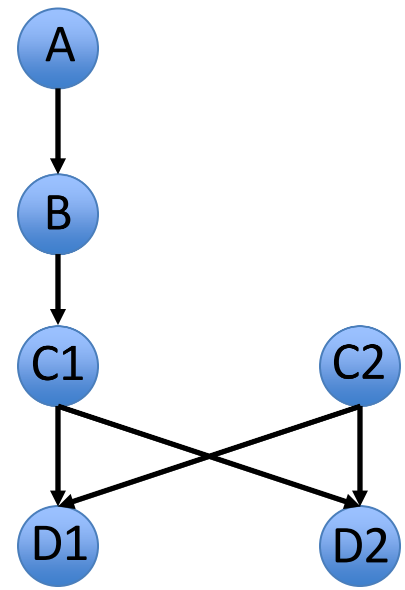
base query: SELECT sup as anc, stu as desc FROM Advisor

UNION → *duplicate eliminating union*

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

Advisor

sup	stu
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B



$Q_R = Anc_0 \bowtie Advisor$

Anc_0

anc	desc

$Ancestor_1 =$

Q_B

anc	desc
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B

∪

Q_R

anc	desc

=

Anc_1

anc	desc
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B

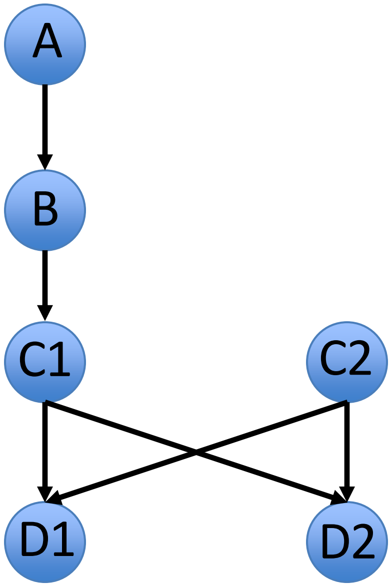
Is $Anc_1 - Ans_0 = \emptyset$?
No: Repeat

TC: ATTEMPT 2: Union w/ a "Base" Case

Anc_1	
anc	desc
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B

→ All 1-degree ancestors

Advisor	
sup	stu
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B



$Q_R = Anc_1 \bowtie Advisor$

$Ancestor_2 =$

Q_B	
anc	desc
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B

∪

Q_R	
anc	desc
A	C1
B	D1

=

Anc_2			
anc	desc	anc	desc
C1	D1	A	C1
C1	D2	B	D1
C2	D1		
C2	D2		
B	C1		
A	B		

Is $Anc_2 – Ans_1 = \emptyset$?
No: Repeat

TC: ATTEMPT 2: Union w/ a "Base" Case

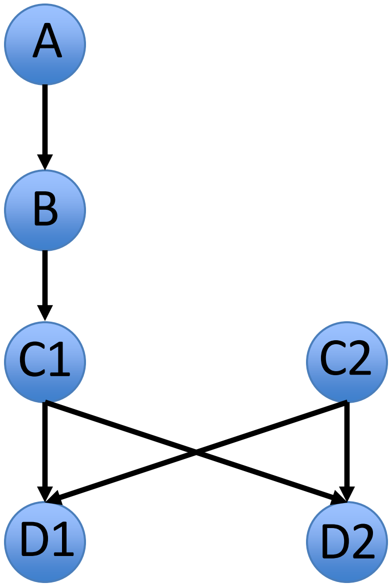
Anc_2

anc	desc	anc	desc
C1	D1	A	C1
C1	D2	B	D1
C2	D1		
C2	D2		
B	C1		
A	B		

→ All 1- and 2-degree ancestors

Advisor

sup	stu
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B



$Q_R = Anc_2 \bowtie Advisor$

$Ancestor_3$ = Q_B ∪ Q_R = Anc_3

Q_B

anc	desc
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B

∪

Q_R

anc	desc
A	C1
B	D1
A	D1
A	D2

=

Anc_3

anc	desc	anc	desc
C1	D1	A	C1
C1	D2	B	D1
C2	D1	A	D1
C2	D2	A	D2
B	C1		
A	B		

Is $Anc_3 – Ans_2 = \emptyset$?
No: Repeat

TC: ATTEMPT 2: Union w/ a “Base” Case

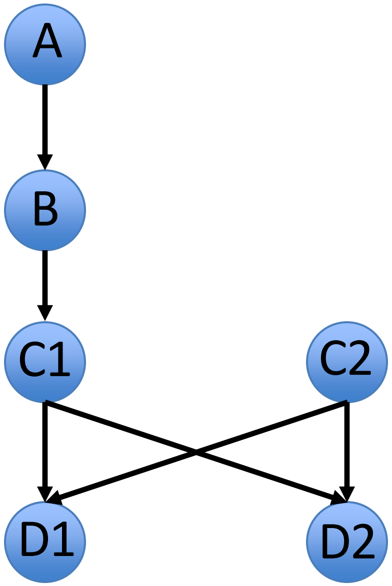
Anc_3

anc	desc	anc	desc
C1	D1	A	C1
C1	D2	B	D1
C2	D1	A	D1
C2	D2	A	D2
B	C1		
A	B		

→ All 1-, 2-, and 3-degree ancestors

Advisor

sup	stu
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B



$Q_R = Anc_3 \bowtie Advisor$

$Ancestor_4$ =

Q_B

anc	desc
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B

∪

Q_R

anc	desc
A	C1
B	D1
A	D1
A	D2

=

Anc_4

anc	desc	anc	desc
C1	D1	A	C1
C1	D2	B	D1
C2	D1	A	D1
C2	D2	A	D2
B	C1		
A	B		

Is $Anc_4 – Ans_3 = \emptyset$?
Yes: Stop

Final Answer called the fixed point of Q.

Some Comments

- Recall common WITH RECURSIVE query template:

  `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 1: Q can still be monotone even if it contains set difference. But not guaranteed to be.
- Note 2: Q can be non-monotone & still converge, i.e. monotonicity is a **sufficient condition** for convergence but **not necessary**

Why Does Monotonicity Guarantee A Unique Fixed Point For A Recursive Query?

- Proof Sketch: Recall fixed point subroutine:

$T_0 = \emptyset$; T_1 = Q (but use T_0 for T); T_2 = Q (but use T_1 for T)

…

- 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 every base relation in Q is finite.

Anc_4			
anc	desc	anc	desc
C1	D1	A	C1
C1	D2	B	D1
C2	D1	A	D1
C2	D2	A	D2
B	C1		
A	B		

Advisor	
sup	stu
C1	D1
C1	D2
C2	D1
C2	D2
B	C1
A	B

- 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)

Example Non-Monotone Recursive Query 1

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

R
x
1
2

T_0
x

T_1
x
1
2

T_2
x
1
2
~~3~~

T_3
x
1
2
~~6~~

T_4
x
1
2
9

… 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 2nd example after we cover "mutual recursion" (stay tuned).

Linear vs Non-linear Recursion

- Recall Q_R in transitive closure:

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

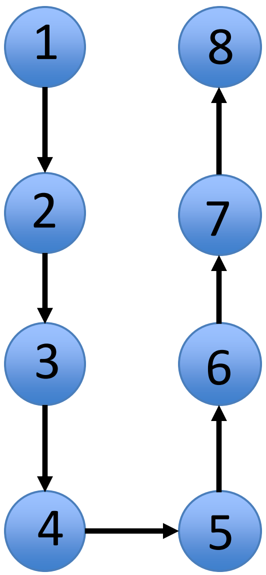
Has 1 reference to itself Ancestor: Called *linear recursion*

Can have > 1 reference to Ancestor, called *non-linear recursion*

Non-linear Recursive Computation of Ancestors

```
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)
```

Advisor	
sup	stu
1	2
2	3
3	4
4	5
5	6
6	7
7	8



Anc_0	
anc	desc

$Ancestor_1$ =

Q_B	
anc	desc
1	2
2	3
3	4
4	5
5	6
6	7
7	8

U

Q_R	
anc	desc

=

Anc_1	
anc	desc
1	2
2	3
3	4
4	5
5	6
6	7
7	8

Is $Anc_1 – Ans_0 = \emptyset$?
No: Repeat

Non-linear Recursive Computation of Ancestors

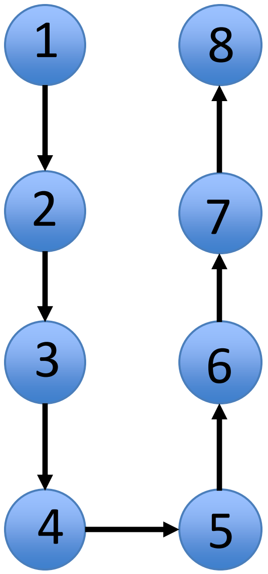
Anc_1

anc	desc
1	2
2	3
3	4
4	5
5	6
6	7
7	8

→ All 1-degree ancestors

Advisor

sup	stu
1	2
2	3
3	4
4	5
5	6
6	7
7	8



Q_B

anc	desc
1	2
2	3
3	4
4	5
5	6
6	7
7	8

∪

Q_R

anc	desc
1	3
2	4
3	5
4	6
5	7
6	8

=

Anc_2

anc	desc	anc	desc
1	2	1	3
2	3	2	4
3	4	3	5
4	5	4	6
5	6	5	7
6	7	6	8
7	8		

Non-linear Recursive Computation of Ancestors

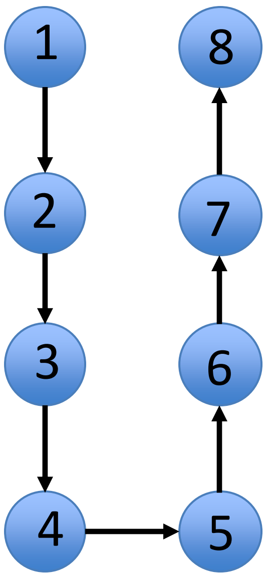
Anc_2

anc	desc	anc	desc
1	2	1	3
2	3	2	4
3	4	3	5
4	5	4	6
5	6	5	7
6	7	6	8
7	8		

→ All 1 and 2-degree ancestors

Advisor

sup	stu
1	2
2	3
3	4
4	5
5	6
6	7
7	8



Q_B

anc	desc
1	2
2	3
3	4
4	5
5	6
6	7
7	8

∪

Q_R

anc	desc	anc	desc	anc	desc
1	3	2	5	4	8
2	4	3	6		
3	5	4	7		
4	6	5	8		
5	7	1	5		
6	8	2	6		
1	4	3	7		

=

Anc_3

a	d	a	d	a	d	a	d
1	2	1	3	2	5	4	8
2	3	2	4	3	6		
3	4	3	5	4	7		
4	5	4	6	5	8		
5	6	5	7	1	5		
6	7	6	8	2	6		
7	8	1	4	3	7		

Non-linear Recursive Computation of Ancestors

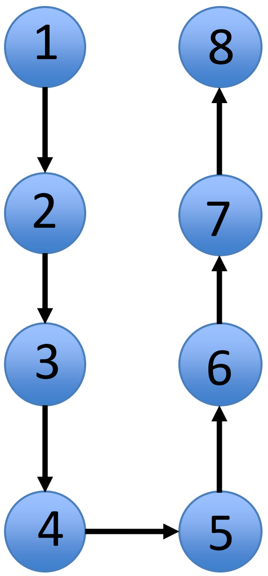
Anc_3

a	d	a	d	a	d	a	d
1	2	1	3	2	5	4	8
2	3	2	4	3	6		
3	4	3	5	4	7		
4	5	4	6	5	8		
5	6	5	7	1	5		
6	7	6	8	2	6		
7	8	1	4	3	7		

→ All 1, 2, 3, and 4-degree ancestors

Advisor

sup	stu
1	2
2	3
3	4
4	5
5	6
6	7
7	8



Q_B

anc	desc
1	2
2	3
3	4
4	5
5	6
6	7
7	8

∪

Q_R

a	d	a	d	a	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

=

Anc_4

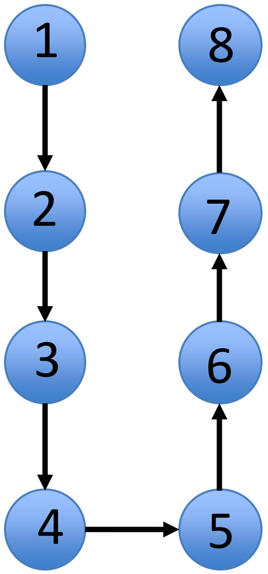
a	d	a	d	a	d	a	d
1	2	1	3	2	5	4	8
2	3	2	4	3	6	1	6
3	4	3	5	4	7	2	7
4	5	4	6	5	8	3	8
5	6	5	7	1	5	1	7
6	7	6	8	2	6	2	8
7	8	1	4	3	7	1	8

Non-linear Recursive Computation of Ancestors

Anc_4							
a	d	a	d	a	d	a	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

→ All 1, … 8-degree ancestors

Advisor	
sup	stu
1	2
2	3
3	4
4	5
5	6
6	7
7	8



Q_B	
anc	desc
1	2
2	3
3	4
4	5
5	6
6	7
7	8

∪

Q_R					
a	d	a	d	a	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

=

Anc_5							
a	d	a	d	a	d	a	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

Is $Anc_5 – Ans_4 = \emptyset$?

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 (ask me after lecture)

Mutual Recursion

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

```
WITH  RECURSIVE
        RECURSIVE R1 AS Q1    →  e.g. references R2
        RECURSIVE R2 AS Q2    →  e.g. references R3
        RECURSIVE R3 AS Q3    →  e.g. references R1
        …
Q
```

- Note: Q1-Q3 may be alone non-recursive but together they may be recursive or they may be recursive alone as well
- So they need to be executed "in tandem" until fixed point.

Mutual Recursion Example

- Table *Natural* (*n*) contains 1,2,3,…
- 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)
```

$Even_0 = \emptyset$, $Odd_0 = \emptyset$

$Even_1 = \emptyset$, $Odd_1 = \{1\}$

$Even_2 = \{2\}$, $Odd_2 = \{1\}$

$Even_3 = \{2\}$, $Odd_3 = \{1, 3\}$

$Even_4 = \{2, 4\}$, $Odd_4 = \{1, 3\}$

$Even_5 = \{2, 4\}$, $Odd_5 = \{1, 3, 5\}$

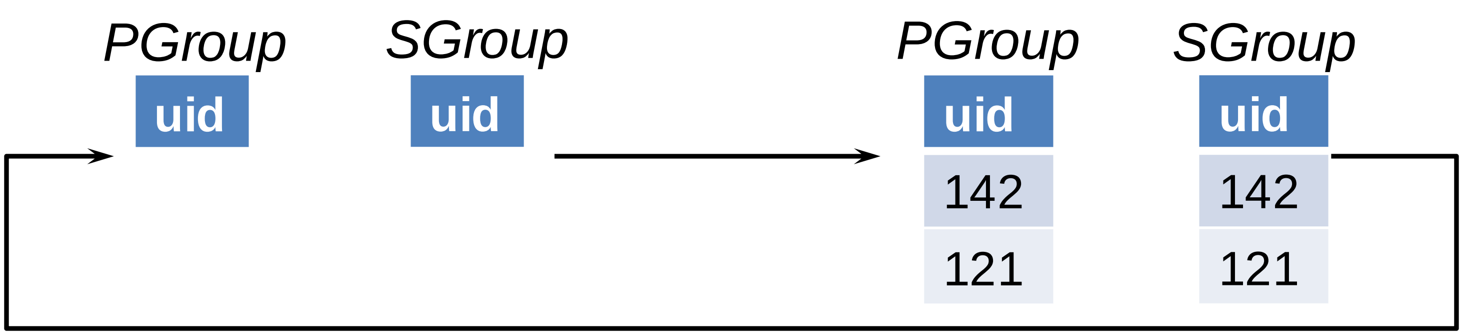
…

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))
```

uid	name	age
142	Bart	10
121	Allison	8

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!

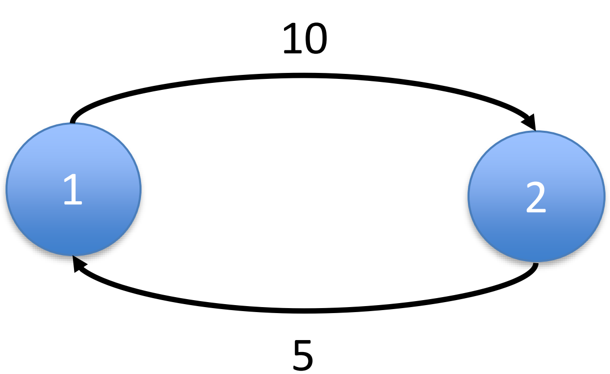
Example Non-convergence Based On Database

- Consider this All Paths query Q:

```
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
    R1 AS Q1
    R2 AS Q2
    …
    Rn AS Qn
```

- 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.

Rest of the slides are optional and included to better understand where SQL recursion is inspired from (which is the Datalog language)

Datalog: Logic-based DB Query Language with Recursion As a First Class Citizen

- A QL based on logical rules of the form: Head := Body
- A DB consists of a set of “base relations” (called “extensional” db)

  Likes(person, foodItem)

  Sells(restaurant, foodItem, cost)

  Frequents(person, restaurant)

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

  Head: Happy(p)

  Body: conjunction/AND of “subgoals”: Likes(p, f), Frequents(p, r), Sells(r, f, c)

- 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: Π_{person} (Likes ⋈ Frequents ⋈ 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
- Similar to context-free grammar rules in programming languages

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),
                             totalCost = cost1 + cost2

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

Very Strong and Beautiful Result

- Given a Datalog program that satisfy some properties (specifically some monotonicity and finiteness rules as before):

```
R1 := body 1 (possibly recursive)
R2 := body 2 (possibly recursive)
…
R2 := body 7 (possibly recursive)
…
Rk := 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 := body 1 500 times if it keeps producing new tuples; then run R_2 := body 2, then Rj, then R_1 again etc.
- Extends the convergence criteria we discussed for SQL recursion

Last Comments On Datalog

- Several DBMSs, e.g., recent RelationalAI, 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)

```
Sibling(x, y) := BioParent(z, x), BioParent(z, y), x != 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