How expressive is SQL?

Full programming languages can express all computable functions (C, Java, etc)

Can SQL express all computable queries?
A: YES B: NO

How expressive is SQL?

flight	from	to
	SD	LA
	SD	ORD
	LA	NY

…………..

Can SQL express the following query:
"Is there a way to get from City1 to City2" ?

A: YES B: NO

Limitation of basic SQL

flight	from	to
	SD	LA
	SD	ORD
	LA	NY
	…………..	

"Is there a way to get from City1 to City2" ?

Easier:

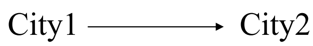
"Is there a way to get from City1 to City2 by a direct flight?"

City1 ⟶ City2

```
Select * from flight
where from = 'City1' and to = 'City2'
```

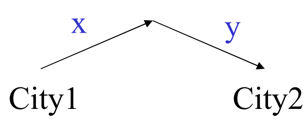
Easier:

"Is there a way to get from City1 to City2 with at most one stopover?"




```
select * from flight
where from = 'City1' and to = 'City2'
```

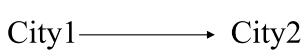
OR




```
select x.from, y.to
from flight x, flight y
where x.from = 'City1' and
x.to = y.from and y.to = 'City2'
```

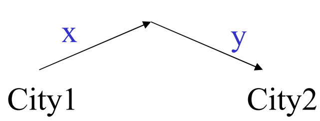
Easier:

"Is there a way to get from City1 to City2 with at most two stopovers?"



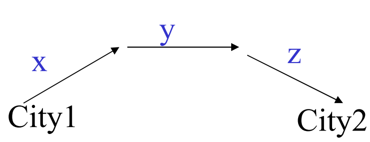

```
select * from flight
where from = 'City1' and to ='City2'
```

OR



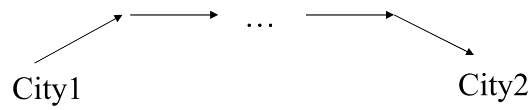

```
select x.from, y.to
from flight x, flight y
where x.from = 'City1' and
x.to = y.from and y.to = 'City2'
```

OR




```
select x.from,  z.to
from flight x, flight y, flight z
where x.from = 'City1' and x.to = y.from
and y.to = z.from and z.to = 'City2'
```

"Is there a way to get from City1 to City2 with at most k stopovers?"



Need k+1 tuple variables!

"Is there a way to get from City1 to City2
with any number of stopovers?"

Cannot do in basic SQL!

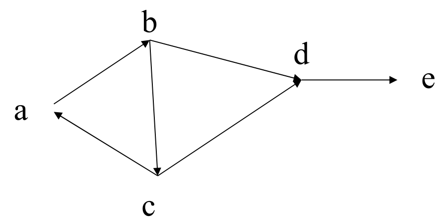
Similar examples

- Parts-components relation:
  "find all subparts of some given part A"
- Parent/child relation
  "find all of John's descendants"

More general: transitive closure of a graph

G	A	B
	a	b
	c	a
	b	c
	………	

Find the pairs of nodes <x,y> that are connected by some directed path



a	b
a	d
a	e
b	d
b	e
……..	

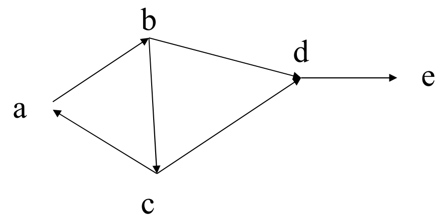
Computing transitive closure T of G

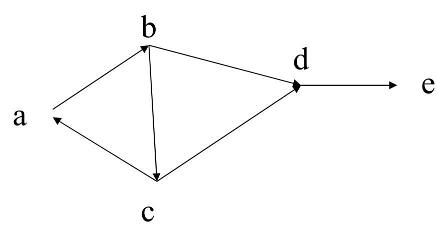
"Find the pairs of nodes <a,b> that are connected in G"

Same as

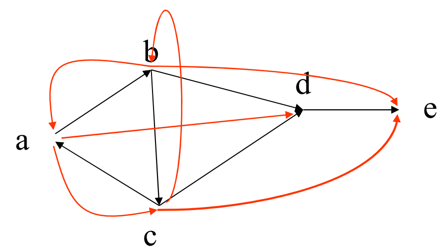
"find pairs of nodes <a,b> at distance 1"   UNION
"find pairs of nodes <a,b> at distance at most 2"   UNION
………..
"find pairs of nodes <a,b> at distance at most k" UNION
………..

When to stop? At some point, no new nodes are added.
Distance cannot be larger than total number of nodes in G.

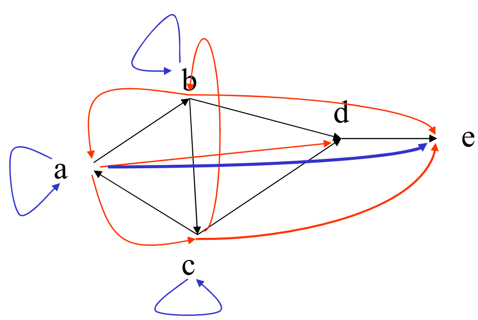




Distance 1



Distance ≤ 2



Distance ≤ 3

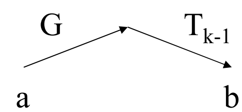
Denote by T_k the pairs of nodes at distance at most k

T_1 : “find pairs of nodes <a,b> at distance 1”

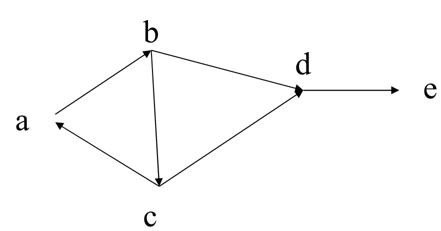
select * from G

T_k : “find the pairs of nodes <a,b> at distance at most k”

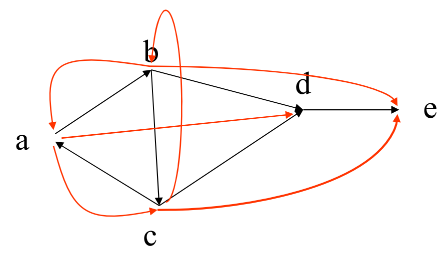
$a \xrightarrow{T_{k-1}} b$ OR



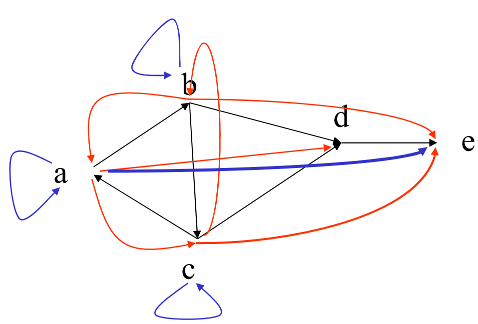
(select * from T_{k-1})
union
(select x.A, y.B from G x, T_{k-1} y
where x.B = y.A)



T_1



T_2



T_3

One solution: extend SQL with recursion
(not part of the standard)

```
create recursive view T as
(select * from G)
union
(select x.A, y.B
from G x, T y
where x.B = y.A)
```

Semantics:
1. Start with empty T
2. While T changes
   {evaluate view with current T;
   union result with T }

Note: this must terminate, since there are finitely many tuples one can add to T (if no new values are created)

Alternative

```
with recursive T as
(select * from G)
union
(select x.A, y.B
from G x, T y
where x.B = y.A)
select * from T ;
```

Semantics:
1. Start with empty T
2. While T changes
   {evaluate view with current T;
   union result with T }

Note: this must terminate, since there are finitely many tuples one can add to T (if no new values are created)

Another example

frequents	drinker bar

Friends: drinkers who frequent the same bar
Find transitive closure of ***Friends***

```
create recursive view T as
select f1.drinker as drinker1, f2.drinker as drinker2
from frequents f1, frequents f2
where f1.bar = f2.bar
union
select t1.drinker1, f2.drinker as drinker 2
from T t1, frequents f1, frequents f2
where t1.drinker2 = f1.drinker and f1.bar = f2.bar
```

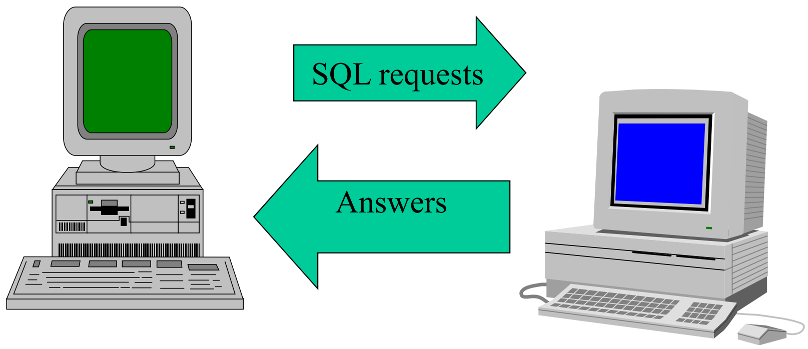
Problematic example

```
create recursive view T as
select  A,  0 as N from R
union
select A, N+1 as N
from T
```

R	A

- never terminates
- some systems forbid arithmetic or aggregate functions in selects in recursive definitions

Powerful way to overcome limitations
embedded SQL



Client: full programming language (Java, C+, etc)

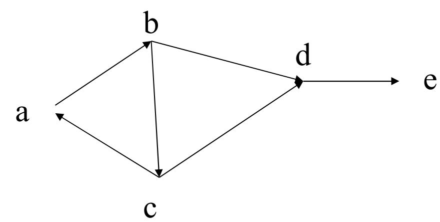
DB Server

23

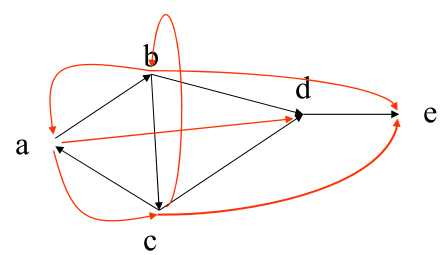
Transitive closure in embedded SQL

<pseudo-code>

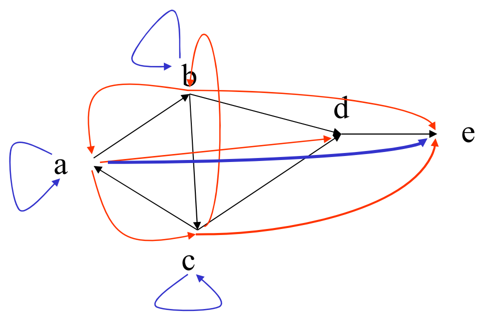
$T := G$
$\Delta := G$
while $\Delta \neq \Phi$ do
{ $T_{old} = T$
T := (select * from T)
union
(select x.A, y.B from G x, T y
where x.B = y.A)
$\Delta := T - T_{old}$ }
Output T



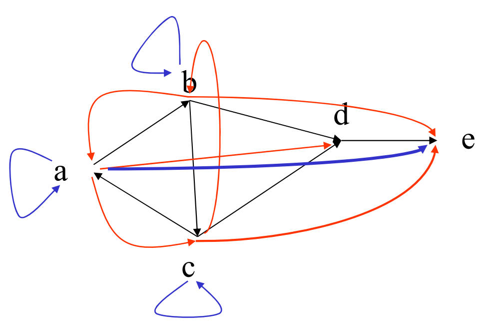
T_1 and Δ_1



T_2 and Δ_2



T_3 and Δ_3



$T_4 = T_3$ and $\Delta_4 = \Phi$

stop!

Another way: Embedded SQL

<pseudo-code>

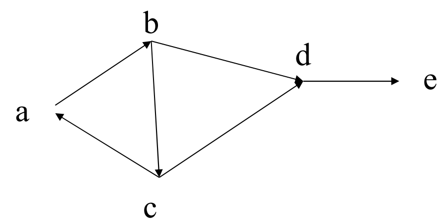
```
T := G
Δ := G
while Δ ≠ Φ do
    { T_old = T
      T := (select * from T)
           union
           (select x.A, y.B from G x, T y
           where x.B = y.A)
      Δ := T − T_old }
Output T
```

Converges in diameter(G) iterations
(maximum distance between two nodes in G)

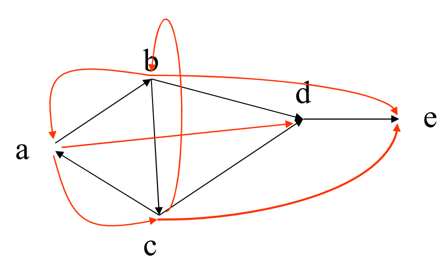
Optimization ("semi-naïve" evaluation): use at least one new tuple (from Δ) every time

<pseudo-code>

```
T := G
Δ := G
while Δ ≠ Φ do
    { T_old = T
      T := (select * from T)
           union
           (select x.A, y.B from G x, Δ y
           where x.B = y.A)
      Δ := T − T_old }
Output T
```

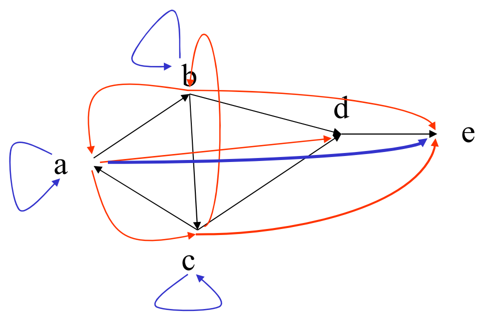


T_1 and Δ_1



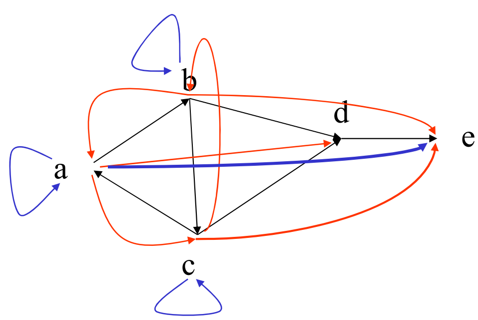
T_2 and Δ_2

No longer recompute <c,b> but recompute <c,d>



T_3 and Δ_3

No longer recompute <a,d> but recompute <c,e>



$T_4 = T_3$ and $\Delta_4 = \Phi$

stop!

Faster convergence (double recursion):

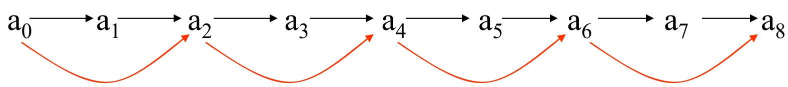
$T := G$
$\Delta := G$
while $\Delta \neq \Phi$ do
  { $T_{old} = T$
  $T :=$ (select * from T)
    union
    (select x.A, y.B from T x, T y
    where x.B = y.A)
  $\Delta := T - T_{old}$ }
Output T

Converges in log(diameter(G)) iterations

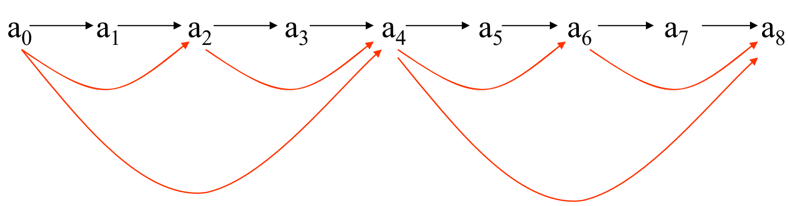
Example (focus on computing $<a_0, a_8>$)

$$a_0 \longrightarrow a_1 \longrightarrow a_2 \longrightarrow a_3 \longrightarrow a_4 \longrightarrow a_5 \longrightarrow a_6 \longrightarrow a_7 \longrightarrow a_8$$

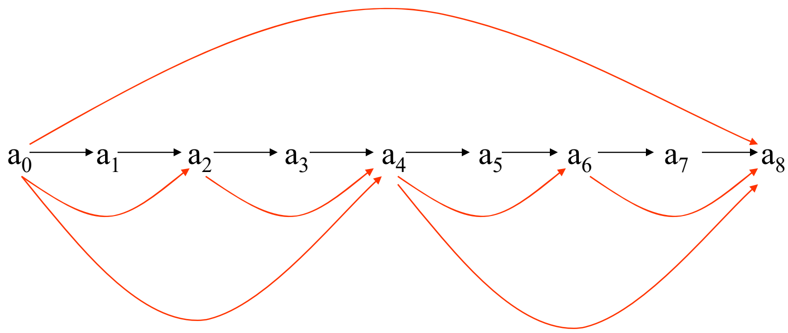
Example (focus on computing $<a_0,a_8>$)



Example (focus on computing $<a_0,a_8>$)



Example (focus on computing $<a_0,a_8>$)



Compare to linear recursion (first program):

$$a_0 \to a_1 \to a_2 \to a_3 \to a_4 \to a_5 \to a_6 \to a_7 \to a_8$$

Compare to linear recursion (first program):

$$a_0 \to a_1 \to a_2 \to a_3 \to a_4 \to a_5 \to a_6 \to a_7 \to a_8$$

Compare to linear recursion (first program):

$$\begin{tikzcd} a_0 \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r] & a_4 \arrow[r] & a_5 \arrow[r] & a_6 \arrow[r] \arrow[rr, bend left] & a_7 \arrow[r] & a_8 \end{tikzcd}$$

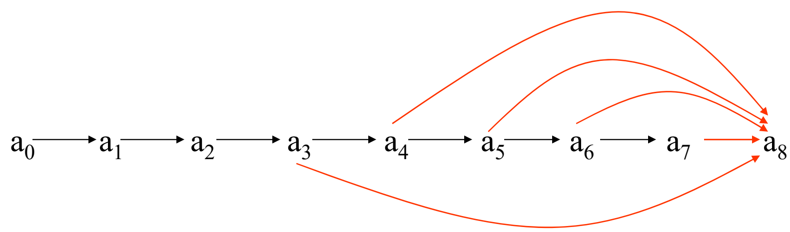
Compare to linear recursion (first program):

$$\begin{tikzcd} a_0 \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r] & a_4 \arrow[r] & a_5 \arrow[r] \arrow[rrr, bend left] & a_6 \arrow[r] \arrow[rr, bend left] & a_7 \arrow[r] & a_8 \end{tikzcd}$$

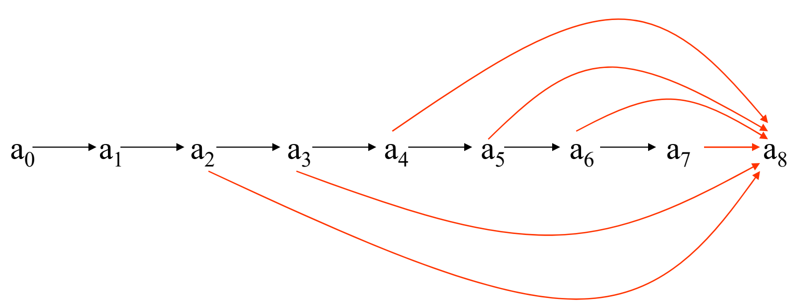
Compare to linear recursion (first program):

$$\begin{tikzcd} a_0 \arrow[r] & a_1 \arrow[r] & a_2 \arrow[r] & a_3 \arrow[r] & a_4 \arrow[r] \arrow[rrrr, bend left] & a_5 \arrow[r] \arrow[rrr, bend left] & a_6 \arrow[r] \arrow[rr, bend left] & a_7 \arrow[r] & a_8 \end{tikzcd}$$

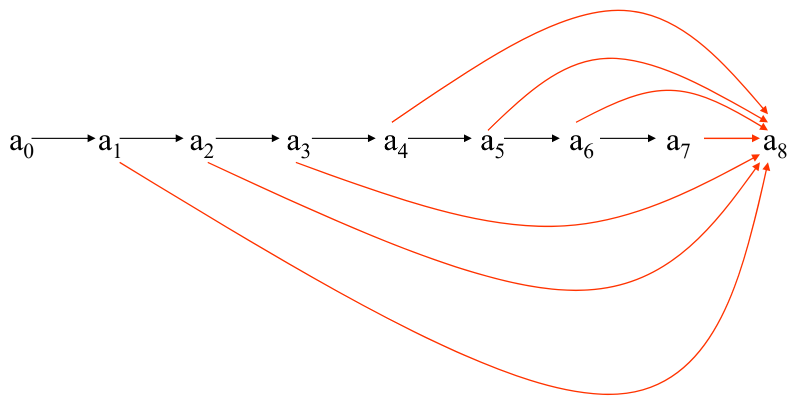
Compare to linear recursion (first program):



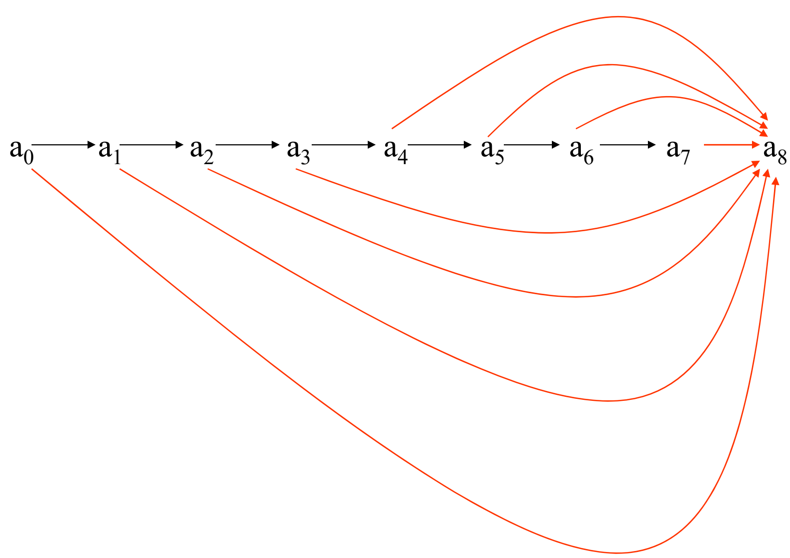
Compare to linear recursion (first program):



Compare to linear recursion (first program):



Compare to linear recursion (first program):



Optimization ("semi-naïve" evaluation):
again, use at least one new tuple every time

```
T := G
Δ := G
while Δ ≠ Φ do
    { T_old = T
      T := (select * from T)
           union
           (select x.A, y.B from Δ x, T y
           where x.B = y.A)
           union
           (select x.A, y.B from T x, Δ y
           where x.B = y.A) }
       Δ := T − T_old
Output T
```