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

<table>
<thead>
<tr>
<th>flight</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>LA</td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>ORD</td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>NY</td>
<td></td>
</tr>
<tr>
<td>........</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

A: YES  B: NO
Limitation of basic SQL

<table>
<thead>
<tr>
<th>flight</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td>LA</td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>ORD</td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>NY</td>
<td></td>
</tr>
</tbody>
</table>

“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?”

\[
\text{select } * \text{ from flight} \\
\text{where from = ‘City1’ and to = ‘City2’}
\]

OR

\[
\text{select x.from, y.to} \\
\text{from flight x, flight y} \\
\text{where x.from = ‘City1’ and} \\
x.to = y.from \text{ and } y.to = ‘City2’
\]

Easier:

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

\[
\text{select } * \text{ from flight} \\
\text{where from = ‘City1’ and to = ‘City2’}
\]

OR

\[
\text{select x.from, y.to} \\
\text{from flight x, flight y} \\
\text{where x.from = ‘City1’ and} \\
x.to = y.from \text{ and } y.to = ‘City2’
\]

OR

\[
\text{select x.from, z.to} \\
\text{from flight x, flight y, flight z} \\
\text{where x.from = ‘City1’ and x.to = y.from} \\
\text{and y.to = z.from and z.to = ‘City2’}
\]
“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

<table>
<thead>
<tr>
<th>G</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

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

```
  a            b        c
  a            b  c  
  a            d  
  b            d  e
  c                
```

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 $k$”

\[
\begin{array}{c}
\text{a} \rightarrow_{T_{k-1}} \text{b} \\
\end{array}
\]

OR

\[
\begin{array}{c}
\text{G} \\
\text{a} \rightarrow_{T_{k-1}} \text{b}
\end{array}
\]

(select * from $T_{k-1}$)

union

(select x.A, y.B from $G$ x, $T_{k-1}$ y
where x.B = y.A)
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 drinker2
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

- never terminates
- some systems forbid arithmetic or aggregate functions in selects in recursive definitions
Transitive closure in embedded SQL

\(<\text{pseudo-code}>\)

\[ \begin{align*}
T & := G \\
\Delta & := G \\
\text{while } \Delta \neq \emptyset \text{ do} \\
\{ & T_{\text{old}} = T \\
& T := \text{(select * from } T) \\
& \text{union} \\
& \text{(select } x.A, y.B \text{ from } G \text{ x, T y} \\
& \text{where } x.B = y.A) \\
& \Delta := T - T_{\text{old}} \}
\end{align*} \]

Output $T$
$T_1$ and $\Delta_1$

$T_2$ and $\Delta_2$
$T_3$ and $\Delta_3$

$T_4 = T_3$ and $\Delta_4 = \Phi$
stop!
Another way: Embedded SQL

<pseudo-code>
T := G
Δ := G
while Δ ≠ Φ do
    { T<sub>old</sub> = T
      T := (select * from T)
      union
       (select x.A, y.B from G x, T y
        where x.B = y.A)
       Δ := T – T<sub>old</sub> }
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<sub>old</sub> = T
      T := (select * from T)
      union
       (select x.A, y.B from G x, Δ y
        where x.B = y.A)
       Δ := T – T<sub>old</sub> }
Output T
\[ T_1 \text{ and } \Delta_1 \]

\[ T_2 \text{ and } \Delta_2 \]

No longer recompute \(<c,b>\) but recompute \(<c,d>\)
No longer recompute \(<a,d>\) but recompute \(<c,e>\)

\[ T_4 = T_3 \text{ and } \Delta_4 = \Phi \]

stop!
Faster convergence (double recursion):

\[
\begin{align*}
T & := G \\
\Delta & := G \\
\text{while } \Delta \neq \emptyset \text{ do} & \\
\{ & T_{\text{old}} = T \\
& T := (\text{select * from } T) \\
& \quad \text{union} \\
& \quad (\text{select } x.A, y.B \text{ from } T \times T) \\
& \quad \text{where } x.B = y.A \\
& \Delta := T - T_{\text{old}} \} \\
\text{Output } T
\end{align*}
\]

Converges in \(\log(\text{diameter}(G))\) iterations

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

\[
\begin{array}{cccccccc}
a_0 & \rightarrow & a_1 & \rightarrow & a_2 & \rightarrow & a_3 & \rightarrow & a_4 \\
\rightarrow & a_5 & \rightarrow & a_6 & \rightarrow & a_7 & \rightarrow & a_8
\end{array}
\]
Example (focus on computing $<a_0,a_8>$)
Example (focus on computing $<a_0,a_8>$)

Compare to linear recursion (first program):

\[ a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \]
Compare to linear recursion (first program):

\[ a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \]
Compare to linear recursion (first program):

\[ a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \]
Compare to linear recursion (first program):

\[ a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \]
Compare to linear recursion (first program):

\[ a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow a_6 \rightarrow a_7 \rightarrow a_8 \]
Optimization (“semi-naïve” evaluation):
again, use at least one new tuple every time

\[
\begin{align*}
T & := G \\
\Delta & := G \\
\text{while } & \Delta \neq \Phi \text{ do} \\
& \{ \\
& \quad T_{\text{old}} = T \\
& \quad T := (\text{select * from } T) \\
& \quad \quad \text{union} \\
& \quad \quad (\text{select } x.A, y.B \text{ from } \Delta x, Ty \\
& \quad \quad \text{where } x.B = y.A) \\
& \quad \quad \text{union} \\
& \quad \quad (\text{select } x.A, y.B \text{ from } Tx, \Delta y \\
& \quad \quad \text{where } x.B = y.A) \} \\
\Delta & := T - T_{\text{old}} \\
\text{Output } T
\end{align*}
\]

JDBC assignment

<table>
<thead>
<tr>
<th>Flight</th>
<th>airline origin</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Connected</th>
<th>airline origin</th>
<th>destination</th>
<th>stops</th>
</tr>
</thead>
</table>