Limitations of SQL

- Reachability queries:

<table>
<thead>
<tr>
<th>Flights</th>
<th>Src</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>'EDI'</td>
<td>'LHR'</td>
<td></td>
</tr>
<tr>
<td>'EDI'</td>
<td>'EWR'</td>
<td></td>
</tr>
<tr>
<td>'EWR'</td>
<td>'LAX'</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Query: Find pairs of cities \((A, B)\) such that one can fly from \(A\) to \(B\) with at most one stop:

\[
\begin{align*}
\text{SELECT } & \text{F1.Src, F2.Dest} \\
\text{FROM } & \text{Flights F1, Flights F2} \\
\text{WHERE } & \text{F1.Dest=F2.Src} \\
\text{UNION} & \\
\text{SELECT } & \text{* FROM Flights}
\end{align*}
\]
Reachability queries cont’d

• Query: Find pairs of cities \((A, B)\) such that one can fly from \(A\) to \(B\) with at most two stops:

\[
\text{SELECT F1.Src, F3.Dest} \\
\text{FROM Flights F1, Flights F2, Flights F3} \\
\text{WHERE F1.Dest=F2.Src AND F2.Dest=F3.Src} \\
\text{UNION} \\
\text{SELECT F1.Src, F2.Dest} \\
\text{FROM Flights F1, Flights F2} \\
\text{WHERE F1.Dest=F2.Src} \\
\text{UNION} \\
\text{SELECT * FROM Flights}
\]
Reachability queries cont’d

• For any fixed number $k$, we can write the query

\[
\text{Find pairs of cities } (A, B) \text{ such that one can fly from } A \text{ to } B \\
\text{with at most } k \text{ stops}
\]

in SQL.

• What about the general reachability query:

\[
\text{Find pairs of cities } (A, B) \text{ such that one can fly from } A \text{ to } B.
\]

• SQL cannot express this query.

• Solution: SQL3 adds a new construct that helps express reachability
queries. (May not yet exist in some products.)
Reachability queries cont’d

• To understand the reachability query, we formulate it as a rule-based query:

\[
reach(x, y) :- \text{flights}(x, y)
\]
\[
reach(x, y) :- \text{flights}(x, z), \ reach(z, y)
\]

• One of these rules is recursive: \(reach\) refers to itself.

• Evaluation:

  - Step 0: \(reach_0\) is initialized as the empty set.
  - Step \(i + 1\): Compute

\[
reach_{i+1}(x, y) :- \text{flights}(x, y)
\]
\[
reach_{i+1}(x, y) :- \text{flights}(x, z), \ reach_i(z, y)
\]

  - Stop condition: If \(reach_{i+1} = reach_i\), then it is the answer to the query.
Evaluation of recursive queries

- Example: assume that flights contains \((a, b), (b, c), (c, d)\).
- Step 0: \(\text{reach} = \emptyset\)
- Step 1: \(\text{reach}\) becomes \(\{(a, b), (b, c), (c, d)\}\).
- Step 2: \(\text{reach}\) becomes \(\{(a, b), (b, c), (c, d), (a, c), (b, d)\}\).
- Step 3: \(\text{reach}\) becomes \(\{(a, b), (b, c), (c, d), (a, c), (b, d), (a, d)\}\).
- Step 4: one attempts to use the rules, but infers no new values for reach. The final answer is thus:

\[\{(a, b), (b, c), (c, d), (a, c), (b, d), (a, d)\}\]
Recursion in SQL3

- SQL3 syntax mimics that of recursive rules:

```sql
WITH RECURSIVE Reach(Src,Dest) AS
(
  SELECT * FROM Flights
  UNION
  SELECT F.Src, R.Dest
  FROM Flights F, Reach R
  WHERE F.Dest=R.Src
)
SELECT * FROM Reach
```
Recursion in SQL3: syntactic restrictions

- There is another way to do reachability as a recursive rule-based query:

  \[
  \text{reach}(x, y) : \neg \text{flights}(x, y) \\
  \text{reach}(x, y) : \neg \text{reach}(x, z), \text{reach}(z, y)
  \]

- This translates into an SQL3 query:

  ```sql
  WITH RECURSIVE Reach(Src,Dest) AS
  (  SELECT * FROM Flights 
      UNION 
      SELECT R1.Src, R2.Dest
      FROM Reach R1, Reach R2 
      WHERE R1.Dest=R2.Src ) 
  SELECT * FROM Reach
  ```

- However, most implementations will disallow this, since they support only \textit{linear} recursion: recursively defined relation is only mentioned once in the \textsc{from} line.
Recursion in SQL3 cont’d

- A slight modification: suppose Flights has another attribute aircraft.
- Query: find cities reachable from Edinburgh.

WITH Cities AS
    SELECT Src, Dest FROM Flights
RECURSIVE Reach(Src, Dest) AS
(
    SELECT * FROM Cities
    UNION
    SELECT C.Src, R.Dest
    FROM Cities C, Reach R
    WHERE C.Dest = R.Src
)

SELECT R.Dest
FROM Reach R
WHERE R.Src = 'EDI'
A note on negation

• Problematic recursion:

WITH RECURSIVE R(A) AS
  (SELECT S.A
   FROM S
   WHERE S.A NOT IN
     SELECT R.A FROM R)

SELECT * FROM R

• Formulated as a rule:

\[ r(x) \iff s(x), \neg r(x) \]
A note on negation cont’d

• Let $s$ contain $\{1, 2\}$.

• Evaluation:
  After step 0: $r_0 = \emptyset$;
  After step 1: $r_1 = \{1, 2\}$;
  After step 2: $r_2 = \emptyset$;
  After step 3: $r_3 = \{1, 2\}$;
  $\ldots$
  After step $2n$: $r_{2n} = \emptyset$;
  After step $2n + 1$: $r_{2n+1} = \{1, 2\}$.

• Problem: it does not terminate!

• What causes this problem? Answer: Negation (that is, NOT IN).
A note on negation cont’d

• Other instances of negation:
  
  EXCEPT
  NOT EXISTS

• SQL3 has a set of very complicated rules that specify when the above operations can be used in WITH RECURSIVE definitions.

• A general rule: it is best to avoid negation in recursive queries.