Extensions of First-Order Logic/Relational Algebra
Practice - Theory

- Much of theory is about First-Order Logic, Relational Algebra, or even subclasses: Conjunctive queries (joins) etc

- Practice: SQL

- Key features:
  - Aggregation (e.g., “average grade per class”)
  - Nulls (missing data)
  - Recursion (reachability queries)
Aggregation

- Extremely common. Over 90% of standard benchmark relational database queries have aggregates. Basis of decision support, data analysis etc.

- Numerical attributes, computation over entire columns

- Adds a lot of power, but still cannot do reachability etc.
Nulls

- Occur everywhere, and mess up everything

- Queries that you expect to work fine just stop working in the presence of nulls

- Bizarre rules, use of many-valued logic (inside SQL) that programmers do not think in terms of
Recursion

- Queries such as reachability in graphs
- Not super common in SQL
- Much more common in theoretical research
- But this is changing thanks for graph databases
- There recursion is the key element of queries, but it is essentially “built in”
Adding counting and aggregation to the language

• Standard SQL feature.
• Assume domain of 2 sorts:
  • usual database entries (graph nodes);
  • numbers (for examples, \( \mathbb{Q} \)).
• Add counting terms and operations:
  • \( \#\bar{x}.\varphi \) – how many \( \bar{x} \) satisfy \( \varphi \).
  • \( P_{\text{property}}(\cdot) \) testing the property of numbers.
• Examples:
  • \( \exists x \ P_{\text{even}}(\#y.E(x, y)) \) – there is a node of even degree.
  • Degree of \( x \) is (degree of \( y \))^2:
    \[ P_{n=m^2}(\#z.E(x, y), \#z.E(y, z)) \]
Adding counting and aggregation to the language

- aggregates and grouping by example: sum up all even degrees in a graph
  - in SQL:  
    ```sql
    SELECT SUM(R.C)
    FROM (SELECT E.A, COUNT(E.B) AS C
          FROM E
          GROUP BY E.A
          HAVING MOD(COUNT(E.B),2) = 0) R
    ```
  - in logic:  
    ```latex
    \text{Aggr}_{SUM} \times (P_{even}(\#y.E(x,y)), \#y.E(x,y))
    ```
Adding counting and aggregation to the language

- aggregates and grouping by example: sum up all even degrees in a graph
  - in SQL: 
    ```
    SELECT SUM(R.C)
    FROM (SELECT E.A, COUNT(E.B) AS C
          FROM E
          GROUP BY E.A
          HAVING MOD(COUNT(E.B),2) = 0) R
    ```
  - in logic: 
    \[ \text{Aggr}_{\text{SUM}} x (\text{P}_{\text{even}}(\#y.E(x,y)), \#y.E(x,y)) \]

- Formally: \( \mathcal{F} \) is an aggregate (e.g., SUM, COUNT...)
- \( \text{aggr\_term}(\bar{x}) = \text{Aggr}_{\mathcal{F}} \bar{y} (\varphi(\bar{x}, \bar{y}), t(\bar{x}, \bar{y})) \)
- Semantics:
  - Find all \( \bar{y}_1, \ldots, \bar{y}_k \) so that \( \varphi(\bar{x}, \bar{y}_i) \) holds
  - Calculate \( v_i = t(\bar{x}, \bar{y}_i) \)
  - \( \text{aggr\_term}(\bar{x}) \) is \( \mathcal{F}(\{v_1, \ldots, v_k\}) \)
Expressiveness of aggregation

- Question: which arithmetic predicates and which aggregate functions to add?
- Let’s be generous: add them all.
- But still look at queries over graph nodes (e.g., transitive closure).

**Theorem**

Queries expressed in the aggregate language with arbitrary arithmetic and aggregates are local: i.e., Hanf-local, Gaifman-local, and have the BNDP.

- In particular, the usual SQL (select-from-where-groupby-having) cannot express transitive closure.
Aggregation and order

• What if we have an order on graph nodes? Can we recover locality?
• No, even in a minimalistic setting:
  • Arithmetic: $<, +, \times$
  • Aggregation: $\text{SUM}$

• If such an aggregate language cannot express transitive closure over ordered graphs, then some complexity classes are separated:
  • $\text{TC}^0$ and $\text{NLOGSPACE}$
  • big open problem in complexity theory
Recursion and Datalog

- Have seen it already:
  - transitive closure:
    
    \[
    \text{trcl}(x, y) \colon= e(x, y) \\
    \text{trcl}(x, y) \colon= e(x, z), \text{trcl}(z, y)
    \]

  - same-generation:
    
    \[
    \text{sg}(x, x) \colon= \\
    \text{sg}(x, y) \colon= e(x', x), e(y', y), \text{sg}(x', y')
    \]

- Now available in the latest SQL standard: \textbf{WITH RECURSIVE}.
  - But without negation.
  - With negation, several semantics exist.
Datalog: expressive power

• Without negation, queries are monotone.
• Even with negation and inflationary semantics:

\begin{center}
\textbf{Theorem (Blass, Kozen, Gurevich, 1985)}

Datalog has the 0-1 law.
\end{center}

• This is \textbf{without order}. What if order is added?
• Then Datalog (with negation) captures \textbf{PTIME}.
• To prove bounds, one needs to separate complexity classes again.
• But without order, it can be separated from \textbf{NP}: 3-colorability is not expressible in Datalog with negation (Dawar, '98).
  • A useful result (recent application in the work on schema mappings)
Extensions: summary

First-Order:
- cannot do recursive (fixed-point) queries;
- cannot count; this continues to hold with the order on the domain.

Extensions:
- with Counting/aggregation:
  - cannot do fixed-point queries
- with fixed-points:
  - cannot count
- But only without ordering on the domain:
  - with ordering, bounds on the as hard as separating complexity classes
Incomplete Information and nulls
The problematic NULL
The problematic NULL
The problematic NULL
The problematic NULL
The problematic NULL

Payment Reminder

Dear [Name],

Your next payment is due on $PAYMENT_DUE_DATE$ for your null null null. Please note, there is an overdue payment on your account. Please go to Account Manager to view details and available services or schedule an online payment.

Ford Credit
accountmanageremail@accountmanageremail.com

[Email]

[Image of Weather Channel]

[Image of Database Table with NULL values]

[Image of Website Sign-in]

Welcome, (null)

Continue as (null)

Select another account
could create lots of trouble for people:

These unlucky people have names that break computers

A few people have names that can utterly confuse the websites they visit, and it makes their life online quite the headache. Why does it happen?

For Null, a full-time mum who lives in southern Virginia in the US, frustrations don’t end with booking plane tickets. She’s also had trouble entering her details into a government tax website, for instance. And when she and her husband tried to get settled in a new city, there were difficulties getting a utility bill set up, too.
And when nulls appear, things go bad
And when nulls appear, things go bad

Textbooks

“fundamentally at odds with the way the world behaves”
“cannot be explained”
And when nulls appear, things go **bad**

**Textbooks**

“fundamentally at odds with the way the world behaves”

“cannot be explained”

**Books for database professionals**

“wrong answers to your queries”

“all results become suspect”

“can never trust the answers”
And when nulls appear, things go **bad**

**Textbooks**

“fundamentally at odds with the way the world behaves”
“cannot be explained”

**Books for database professionals**

“wrong answers to your queries”
“all results become suspect”
“can never trust the answers”

**News headlines**

“Leeds children's heart surgery halted by 'incomplete' data”
“non-existent bills because the companies have incomplete information”
What we have now

**THEORY:**
correctness, but at a **huge** cost

**PRACTICE:**
efficiency, but correctness sacrificed

**Correctness:** certain answers
to be defined soon...

Just run queries and hope for the best...
even more than “just run”: use a many-valued logic...
Incomplete data and certain answers

Incomplete database $D$ represents many complete databases $D_1, D_2, \ldots$

This is done by interpreting incompleteness

For example, by assigning values to every null that occurs in $D$
Incomplete data and certain answers

Tuple \( a \) is certain answer to query \( Q \) in \( D \)
\[ \Leftrightarrow \ a \text{ is an answer to } Q \text{ in every } D_i \]

Certainty is hard computationally: \( \text{coNP-hard} \) for relational algebra (first-order logic) queries
Marked nulls - common in data integration, exchange, OBDA, generalize SQL nulls

Valuations $v$: Nulls $\rightarrow$ Constants

The model
Valuations are homomorphisms

- Database elements come from two sets:
  - **constants** (numbers, strings, etc)
  - **nulls**, denoted by \( \bot_1 \bot_2 \bot_3 \ldots \)

- Homomorphisms
  - \( h(c) = c \) for constants, \( h(\bot) \) is a constant or null
  - valuations \( v \): in addition, \( v(\bot) \) is always a constant

- \( \lbrack D \rbrack = \{ v(D) \mid v \text{ is a valuation} \} \)
Certain Answers
Certain Answers

For Boolean queries: $Q$ is certainly true in $D \iff Q$ is true in $[D]$ - that is, true in $v(D)$ for each valuation $v$
 Certain Answers

For Boolean queries: \(Q\) is certainly true in \(D\) ⇔
\(Q\) is true in \([D]\) - that is, true in \(v(D)\) for each valuation \(v\)

For queries returning tuples, for tuples of constants:
c is a certain answer ⇔ \(c \in Q(v(D))\) for each valuation \(v\)
Certain Answers

For Boolean queries: \( Q \) is certainly true in \( D \) \( \iff \)
\( Q \) is true in \( [D] \) - that is, true in \( v(D) \) for each valuation \( v \)

For queries returning tuples, for tuples of constants:
\( c \) is a certain answer \( \iff c \in Q(v(D)) \) for each valuation \( v \)

An arbitrary tuple \( a \) is a certain answer \( \iff \)
\( v(a) \in Q(v(D)) \) for each valuation \( v \)
Certain answers are coNP-complete for first-order queries

- Boolean $Q$. Certainty is in coNP: Guess a valuation $v$ so that $Q$ is false in $v(D)$.

- Hardness for unions of CQ with negation. Take a graph $G$ with nodes $N$ and edges $E$.

  - For each node $n \in N$, create a new null $\perp_n$. For an edge $(n,n')$, put $(\perp_n, \perp_{n'})$ in $E$.

- Query $Q$: $\exists x \ E(x,x) \lor \exists x,y,z,u \ (x,y,z,u \text{ are different})$

- $Q$ is certainly true iff the graph is not 3-colorable
### Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

### Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>ord2</td>
</tr>
</tbody>
</table>
### Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

### Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>ord2</td>
</tr>
</tbody>
</table>

**Query:** Unpaid orders

**Answer:** ord3
<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Query

Unpaid orders
Unpaid orders
Incomplete databases

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Incomplete databases

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>value1</td>
</tr>
</tbody>
</table>
Incomplete databases

**Orders**

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

**Payments**

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Orders**

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

**Payments**

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>value2</td>
</tr>
</tbody>
</table>
Incomplete databases

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>value3</td>
</tr>
</tbody>
</table>
Incomplete databases

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

D

D₁ → D₂ → D₃ → ... → Dₙ
Incomplete databases

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Possible worlds represented by D
Querying incomplete databases

Certain answers:

Answers that are true in all possible worlds

**Orders**

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

**Payments**

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Query

Unpaid orders
Querying incomplete databases

Certain answers:

Answers that are true in all possible worlds

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Query

Unpaid orders

Certain answers

Ø
Querying incomplete databases

Unpaid orders:

SELECT O.num FROM Orders O WHERE NOT EXISTS (SELECT * FROM Payments P WHERE P.ord = O.num)

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Query

Unpaid orders
Querying incomplete databases

Unpaid orders:

```
SELECT O.num FROM Orders O WHERE NOT EXISTS ( 
    SELECT * FROM Payments P WHERE P.ord = O.num )
```

Orders

<table>
<thead>
<tr>
<th>num</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>ord1</td>
<td>100</td>
</tr>
<tr>
<td>ord2</td>
<td>150</td>
</tr>
<tr>
<td>ord3</td>
<td>135</td>
</tr>
</tbody>
</table>

Payments

<table>
<thead>
<tr>
<th>pid</th>
<th>ord</th>
</tr>
</thead>
<tbody>
<tr>
<td>pay1</td>
<td>ord1</td>
</tr>
<tr>
<td>pay2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Certain answers

ord2, ord3
Are wrong answers common in SQL?

Experiment on the TPC-H Benchmark: models a business scenario with associated decision support queries

(from Guagliardo/L., PODS’16)
# A company database: orders, customers, payments

## Orders

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

## Pay

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

## Customer

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical queries, as we teach students to write them:
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>&quot;Big Data&quot;</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>&quot;SQL&quot;</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>&quot;Logic&quot;</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical queries, as we teach students to write them:

**Unpaid orders:**

```sql
select O.order_id
from Orders O
where O.order_id not in
  (select order from Pay P)
```
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical queries, as we teach students to write them:

**Unpaid orders:**
select O.order_id
from Orders O
where O.order_id not in
  (select order from Pay P)

**Customers without an order:**
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
   where C.cust_id=P.cust_id
   and P.order=O.order_id)
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical queries, as we teach students to write them:

**Unpaid orders:**

select O.order_id
from Orders O
where O.order_id not in
    (select order from Pay P)

**Customers without an order:**

select C.cust_id from Customer C
where not exists
    (select * from Orders O, Pay P
     where C.cust_id=P.cust_id
     and P.order=O.order_id)

Answer: Ord3.

Answer: none.
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

Typical queries, as we teach students to write them:

**Unpaid orders:**

```sql
select O.order_id
from Orders O
where O.order_id not in
  (select order from Pay P)
```

**Customers without an order:**

```sql
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
   where C.cust_id=P.cust_id
   and P.order=O.order_id)
```
### A company database: orders, customers, payments

#### Orders

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

#### Pay

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

#### Customer

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

#### Unpaid orders:

```
select O.order_id
from Orders O
where O.order_id not in
  (select order from Pay P)
```

#### Customers without an order:

```
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
   where C.cust_id=P.cust_id
   and P.order=O.order_id)
```
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>&quot;Big Data&quot;</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>&quot;SQL&quot;</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>&quot;Logic&quot;</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>Ord2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

In the real world, information is often missing

**Unpaid orders:**

```sql
select O.order_id
from Orders O
where O.order_id not in
  (select order from Pay P)
```

**Customers without an order:**

```sql
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
   where C.cust_id=P.cust_id
   and P.order=O.order_id)
```
A company database: orders, customers, payments

Orders

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

Pay

Customer

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

In the real world, information is often missing

Unpaid orders:
select O.order_id
from Orders O
where O.order_id not in
  (select order from Pay P)

Customers without an order:
select C.cust_id from Customer C
where not exists
  (select * from Orders O, Pay P
    where C.cust_id=P.cust_id
    and P.order=O.order_id)
A company database: orders, customers, payments

Orders

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

Pay

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>null</td>
</tr>
</tbody>
</table>

Customer

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

In the real world, information is often missing

**Unpaid orders:**

select O.order_id  
from Orders O  
where O.order_id not in  
(select order from Pay P)

**Customers without an order:**

select C.cust_id from Customer C  
where not exists  
(select * from Orders O, Pay P  
where C.cust_id=P.cust_id  
and P.order=O.order_id)
A company database: orders, customers, payments

<table>
<thead>
<tr>
<th>ORDER_ID</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ord1</td>
<td>“Big Data”</td>
<td>30</td>
</tr>
<tr>
<td>Ord2</td>
<td>“SQL”</td>
<td>35</td>
</tr>
<tr>
<td>Ord3</td>
<td>“Logic”</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Ord1</td>
</tr>
<tr>
<td>c2</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CUST_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
</tr>
<tr>
<td>c2</td>
<td>Mary</td>
</tr>
</tbody>
</table>

In the real world, information is often missing

Unpaid orders:
select O.order_id
from Orders O
where O.order_id not in
 (select order from Pay P)

Customers without an order:
select C.cust_id from Customer C
where not exists
 (select * from Orders O, Pay P
 where C.cust_id=P.cust_id
 and P.order=O.order_id)

Old Answer: Ord3      New: NONE!      Old answer: none      New: c2!
What’s the deal with nulls?

- Back in the 1980s, when SQL was standardized, it chose a 3-valued logic for handling nulls
  - truth values: t, f, u 
  - conditions such as 1 = null evaluate to u
  - propagated using Kleene’s logic:

<table>
<thead>
<tr>
<th></th>
<th>¬</th>
<th>Λ</th>
<th>t</th>
<th>f</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>f</td>
<td>t</td>
<td>t</td>
<td>f</td>
<td>u</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>u</td>
<td>u</td>
<td>f</td>
<td>u</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>∨</th>
<th>t</th>
<th>f</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>f</td>
<td>t</td>
<td>f</td>
<td>f</td>
<td>u</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>f</td>
<td>u</td>
<td>u</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>u</td>
<td>u</td>
<td>u</td>
</tr>
</tbody>
</table>
SQL and 3VL (3-valued logic)

- Constant source of confusion for programmers
- Committee design, just to handle nulls
- Heavily criticized ever since
- One more example of confusion.
Compute $R - S$

Q1:
```
SELECT R.A FROM R
EXCEPT
SELECT S.A FROM S
```

Q2:
```
SELECT R.A FROM R
WHERE R.A NOT IN (SELECT S.A FROM S)
```

Q3:
```
SELECT R.A FROM R
WHERE NOT EXISTS (SELECT S.A FROM S
WHERE S.A=R.A)
```
Compute $R - S$

Answer

$$Q_1 = \text{SELECT} \ R.A \ \text{FROM} \ R \ \text{EXCEPT} \ \text{SELECT} \ S.A \ \text{FROM} \ S$$

$$Q_2 = \text{SELECT} \ R.A \ \text{FROM} \ R \ \text{WHERE} \ R.A \ \text{NOT IN} \ ( \ \text{SELECT} \ S.A \ \text{FROM} \ S )$$

$$Q_3 = \text{SELECT} \ R.A \ \text{FROM} \ R \ \text{WHERE} \ \text{NOT EXISTS} \ ( \ \text{SELECT} \ S.A \ \text{FROM} \ S \ \text{WHERE} \ S.A = R.A )$$

$A$

1
Compute $R - S$

Answer

Q₁

\[
\text{SELECT } R.\text{A FROM } R \\
\text{EXCEPT} \\
\text{SELECT } S.\text{A FROM } S
\]

Q₂

\[
\text{SELECT } R.\text{A FROM } R \\
\text{WHERE } R.\text{A NOT IN} ( \\
\text{SELECT } S.\text{A FROM } S )
\]

Q₃

\[
\text{SELECT } R.\text{A FROM } R \\
\text{WHERE NOT EXISTS} ( \\
\text{SELECT } S.\text{A FROM } S \\
\text{WHERE } S.\text{A} = R.\text{A} )
\]
Compute $R - S$

**Q1**
SELECT R.A FROM R
EXCEPT
SELECT S.A FROM S

**Q2**
SELECT R.A FROM R
WHERE R.A NOT IN (SELECT S.A FROM S)

**Q3**
SELECT R.A FROM R
WHERE NOT EXISTS (SELECT S.A FROM S WHERE S.A=R.A)
Compute \( R - S \)

Answer

Q_1

\[
\text{SELECT } R.A \text{ FROM } R
\]

\[
\text{EXCEPT}
\]

\[
\text{SELECT } S.A \text{ FROM } S
\]

Q_2

\[
\text{SELECT } R.A \text{ FROM } R
\]

\[
\text{WHERE } R.A \text{ NOT IN (}
\]

\[
\text{SELECT } S.A \text{ FROM } S
\]

Q_3

\[
\text{SELECT } R.A \text{ FROM } R
\]

\[
\text{WHERE NOT EXISTS (}
\]

\[
\text{SELECT } S.A \text{ FROM } S
\]

\[
\text{WHERE S.A=} R.A
\]
Compute \( R - S \)

**Answer**

\[
\text{SELECT R.A FROM R EXCEPT SELECT S.A FROM S}
\]

\[
\text{SELECT R.A FROM R WHERE R.A NOT IN ( SELECT S.A FROM S )}
\]

\[
\text{SELECT R.A FROM R WHERE NOT EXISTS ( SELECT S.A FROM S WHERE S.A=R.A )}
\]
Compute $R - S$

Answer

Q₁

$\text{SELECT } R.A \text{ FROM } R$

EXCEPT

$\text{SELECT } S.A \text{ FROM } S$

Q₂

$\text{SELECT } R.A \text{ FROM } R$

WHERE $R.A$ NOT IN ( 

$\text{SELECT } S.A \text{ FROM } S$

)

Q₃

$\text{SELECT } R.A \text{ FROM } R$

WHERE NOT EXISTS ( 

$\text{SELECT } S.A \text{ FROM } S$

WHERE $S.A=R.A$

)
Compute $R - S$

Answer

Q₁
SELECT R.A FROM R EXCEPT SELECT S.A FROM S

Q₂
SELECT R.A FROM R WHERE R.A NOT IN ( SELECT S.A FROM S )

Q₃
SELECT R.A FROM R WHERE NOT EXISTS ( SELECT S.A FROM S WHERE S.A = R.A )
Logic of nulls in SQL…

• A single null value

• 2-valued semantics for $R(a)$, SQL semantics for $(a=b)$
How to interpret atoms?

Standard 2-valued semantics: $$R(a) = \begin{cases} t & \text{if } a \in R \\ f & \text{if } a \not\in R \end{cases}$$

SQL semantics: $$(a=b) = \begin{cases} t & \text{if } a, b \neq \text{NULL and } a=b \\ f & \text{if } a, b \neq \text{NULL and } a \neq b \\ u & \text{if } a \text{ or } b \text{ is NULL} \end{cases}$$
A logician’s approach

- First Order Logic (FO)
  - domain has usual values and NULL
  - Syntactic equality: NULL = NULL but NULL ≠ 1 etc
  - Boolean logic rules for ∧, ∨, ¬
  - Quantifiers: ∀ is conjunction, ∃ is disjunction
- Why would one even think of anything else??
What did SQL do?

- 3-valued FO (a textbook version)
  - domain has usual values and NULL
  - comparisons with NULL result in unknown
  - Kleene logic rules for $\land$, $\lor$, $\neg$
  - Quantifiers: $\forall$ is conjunction, $\exists$ is disjunction
What did SQL do?

- 3-valued FO (a textbook version)
  - domain has usual values and **NULL**
  - comparisons with **NULL** result in **unknown**
  - Kleene logic rules for ∧, ∨, ¬
  - Quantifiers: ∀ is conjunction, ∃ is disjunction

- Seemingly more expressive.
What did SQL do?

• 3-valued FO (a textbook version)
  • domain has usual values and NULL
  • comparisons with NULL result in unknown
  • Kleene logic rules for $\land$, $\lor$, $\neg$
  • Quantifiers: $\forall$ is conjunction, $\exists$ is disjunction

• Seemingly more expressive.

• But does it correspond to reality?
SQL logic is **NOT** 2-valued or 3-valued: it’s a **mix**

- Conditions in **WHERE** are evaluated under 3-valued logic. But then only those evaluated to **true** matter.

- Studied before for **propositional** logic:
  
  - In 1939, Russian logician Bochvar wanted to give a formal treatment of logical paradoxes. To assert that something is true, he introduced a new connective: \( \uparrow p \) means that \( p \) is true.

- Amazingly, 40 years later SQL adopted the same idea.
What did SQL really do?

• 3-valued FO with \( \uparrow \):

  • As textbook version but with the extra connective \( \uparrow \)

\[
\uparrow \varphi = \begin{cases} 
  t, & \text{if } \varphi \text{ is } t \\
  f, & \text{if } \varphi \text{ is } f \text{ or } u
\end{cases}
\]
What *is* the logic of SQL?
What *is* the logic of SQL?

- We have:
  - logician’s 2-valued FO
  - 3-valued FO (Kleene logic)
  - 3-valued FO + Bochvar’s assertion (SQL logic)
What *is* the logic of SQL?

- We have:
  - logician’s 2-valued FO
  - 3-valued FO (Kleene logic)
  - 3-valued FO + Bochvar’s assertion (SQL logic)

- AND THEY ARE ALL THE SAME IN TERMS OF EXPRESSIVE POWER
Bottom line

• SQL will continue to operate with the 3-valued logic

• If you database may have nulls, and most do, you have to keep it in mind when writing queries.
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:

  ```
  SELECT F1.Src, F2.Dest
  FROM Flights F1, Flights F2
  WHERE F1.Dest=F2.Src
  UNION
  SELECT * FROM Flights
  ```
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

  Find pairs of cities $(A, B)$ such that one can fly from $A$ to $B$
  with at most $k$ stops

  in SQL.

• What about the general reachability query:

  Find pairs of cities $(A, B)$ such that one can fly from $A$ to $B$.

• SQL cannot express this query.

• Solution: from SQL 1999, a new construct is added that helps express
  reachability queries. (Does not exist in all products.)
Reachability queries cont’d

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

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

• One of these rules is recursive: \text{reach} refers to itself.

• Evaluation:

  - Step 0: \( \text{reach}_0 \) is initialized as the empty set.
  - Step \( i + 1 \): Compute

\[
\text{reach}_{i+1}(x, y) \leftarrow \text{flights}(x, y)
\]
\[
\text{reach}_{i+1}(x, y) \leftarrow \text{flights}(x, z), \text{reach}_i(z, y)
\]

  - Stop condition: If \( \text{reach}_{i+1} = \text{reach}_i \), 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: \(reach = \emptyset\)

• Step 1: \(reach\) becomes \(\{(a, b), (b, c), (c, d)\}\).

• Step 2: \(reach\) becomes \(\{(a, b), (b, c), (c, d), (a, c), (b, d)\}\).

• Step 3: \(reach\) 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:

\[
\{(a, b), (b, c), (c, d), (a, c), (b, d), (a, d)\}
\]
Datalog

• That was an example of a Datalog program
• Datalog program consists of rules

\[ S_0(\bar{x}) \leftarrow S_1(\bar{u}_1), \ldots, S_k(\bar{u}_k), R_1(\bar{v}_1), \ldots, R_m(\bar{v}_m) \]

• \( R_1, \ldots, R_k \) are among database relations
• Relations \( S_0, S_1, \ldots \) that appear on the left of the rules are called idb (intensional database) predicates
• Semantics: least fixed point
Datalog semantics

- Rule: $S_0(\bar{x}) :– S_1(\bar{u}_1), \ldots, S_k(\bar{u}_k), R_1(\bar{v}_1), \ldots, R_m(\bar{v}_m)$
- Start with all intensional predicates being empty, $S_0^0, S_1^0, S_2^0, \ldots = \emptyset$
- Let $\bar{y}$ be variables in $\bar{u}_1, \ldots, \bar{u}_k, \bar{v}_1, \ldots, \bar{v}_m$ except those in $\bar{x}$
- $S_{n+1}^0$ is the result of evaluation of
  \[
  \exists \bar{y} \ (S_1^m(\bar{u}_1) \land \cdots \land S_k^m(\bar{u}_k) \land R_1(\bar{v}_1) \land \cdots \land R_m(\bar{v}_m))
  \]
  or
  \[
  \pi_{\bar{x}}(S_1^m(\bar{u}_1) \Join \cdots \Join S_k^m(\bar{u}_k) \Join R_1(\bar{v}_1) \Join \cdots \Join R_m(\bar{v}_m))
  \]
- Evaluation stops when the next stage does not add any tuples: $S_{i+1}^n = S_i^n$ for all $S_i$
Recursion in SQL

- SQL 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 SQL 1999: syntactic restrictions

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

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

• This translates into an SQL 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 FROM line.
Recursion in SQL 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:

\[ r(x) :\neg s(x), \neg r(x) \]
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: $r_3 = \{1, 2\}$;
  
  \[
  \vdots
  \]
  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 $\text{IN}$).
A note on negation cont’d

• Other instances of negation:
  
  EXCEPT
  NOT EXISTS

• SQL has a set of syntactic 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.