By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race.

— Alfred North Whitehead (1861 - 1947)

Relational Query Languages

- Query languages: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages != programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
- **Relational Algebra**: More operational, very useful for representing execution plans.
- **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-procedural, declarative.)

⇒ Understanding Algebra & Calculus is key to understanding SQL, query processing!

Preliminaries

- A query is applied to relation instances, and the result of a query is also a relation instance.
  - Schemas of input relations for a query are fixed (but query will run over any legal instance)
  - The schema for the result of a given query is also fixed. It is determined by the definitions of the query language constructs.
- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL
  - Though positional notation is not encouraged

Relational Algebra: 5 Basic Operations

- **Selection** (σ): Selects a subset of rows from relation (horizontal).
- **Projection** (π): Retains only wanted columns from relation (vertical).
- **Cross-product** (×): Allows us to combine two relations.
- **Set-difference** (−): Tuples in r1, but not in r2.
- **Union** (∪): Tuples in r1 or in r2.

Since each operation returns a relation, operations can be composed! (Algebra is "closed").

Example Instances

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>guppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>
The Projection ($\pi$) operation preserves only those attributes that are in the projection list.

- **Examples:** 
  - $\pi_{age}(S)$
  - $\pi_{sname, rating}(S)$

- **Schema** of result:
  - Exactly the fields in the projection list, with the same names as they had in the input relation.

- Projection operator has to eliminate duplicates (How do they arise? Why remove them?)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

Selection ($\sigma$)

- Selects rows that satisfy selection condition.
- Result is a relation.
- Schema of result is same as that of the input relation.
- Do we need to do duplicate elimination?

Union and Set-Difference

- Both of these operations take two input relations, which must be union-compatible:
  - Same number of fields.
  - Corresponding fields have the same type.

- For which, if any, is duplicate elimination required?
Cross-Product
• $S_1 \times R_1$: Each row of $S_1$ paired with each row of $R_1$.
• Q: How many rows in the result?
  • Result schema has one field per field of $S_1$ and $R_1$, with field names ‘inherited’ if possible.
    – May have a naming conflict: Both $S_1$ and $R_1$ have a field with the same name.
    – In this case, can use the renaming operator:
      $\rho (C(l \to s_1, 5 \to s_2), S_1 \times R_1)$

Cross Product Example

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

Compound Operator: Intersection
• In addition to the 5 basic operators, there are several additional “Compound Operators”
  – These add no computational power to the language, but are useful shorthands.
  – Can be expressed solely with the basic ops.
• Intersection takes two input relations, which must be union-compatible.
• Q: How to express it using basic operators?
  $R \cap S = R - (R - S)$

Intersection

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

Compound Operator: Join
• Joins are compound operators involving cross product, selection, and (sometimes) projection.
• Most common type of join is a “natural join” (often just called “join”). $R \bowtie S$ conceptually is:
  – Compute $R \times S$
  – Select rows where attributes that appear in both relations have equal values
  – Project all unique attributes and one copy of each of the common ones.
• Note: Usually done much more efficiently than this.

Natural Join Example

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>45.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

$S_1 \bowtie R_1 = S_1 \cap S_2$
Other Types of Joins

• **Condition Join** (or "theta-join"): 
  \[ R \bowtie_c S = \sigma_c (R \times S) \]
  
  - Result schema same as that of cross-product.
  - May have fewer tuples than cross-product.

  **Equi-Join**: Special case: condition \( c \) contains only conjunction of equalities.

  \[
  \begin{array}{|c|c|c|c|c|}
  \hline
  \text{(sid)} & \text{sname} & \text{rating} & \text{age} & \text{bid} & \text{day} \\
  \hline
  22 & dustin & 7 & 45.0 & 103 & 11/12/96 \\
  31 & lubber & 8 & 55.5 & 103 & 11/12/96 \\
  \hline
  \end{array}
  \]

  \[ S \bowtie_c S \mid \text{sid} < R \text{.sid} \]

  - Result schema same as that of cross-product.
  - May have fewer tuples than cross-product.


Examples of Division \( A/B \)

Q for intuition: What is \( (R/S) \times S \)?

<table>
<thead>
<tr>
<th>( A )</th>
<th>( A/B1 )</th>
<th>( A/B2 )</th>
<th>( A/B3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

**Examples**

**Reserves**

\[
\begin{array}{|c|c|c|}
\hline
\text{sid} & \text{bid} & \text{day} \\
\hline
22 & 101 & 10/10/96 \\
58 & 103 & 11/12/96 \\
\hline
\end{array}
\]

**Sailors**

\[
\begin{array}{|c|c|c|}
\hline
\text{sid} & \text{sname} & \text{rating} & \text{age} \\
\hline
22 & dustin & 7 & 45.0 \\
31 & lubber & 8 & 55.5 \\
58 & rusty & 10 & 35.0 \\
\hline
\end{array}
\]

**Boats**

\[
\begin{array}{|c|c|c|}
\hline
\text{bid} & \text{bname} & \text{color} \\
\hline
101 & Interlake & Blue \\
102 & Interlake & Red \\
103 & Clipper & Green \\
104 & Marine & Red \\
\hline
\end{array}
\]

**Compound Operator: Division**

• Useful for expressing "for all" queries like:
  Find sids of sailors who have reserved all boats.
  - For \( A/B \), attributes of \( B \) must be subset of all attrs of \( A \).
  - May need to "project" to make this happen.
  - E.g., let \( A \) have 2 fields, \( x \) and \( y \); \( B \) have only field \( y \):
    \[
    A/B \text{ contains all tuples } (x) \text{ such that for every } y \text{ tuple in } B, \text{ there is an } xy \text{ tuple in } A.
    \]

**Expressing \( A/B \) Using Basic Operators**

• Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially.)

  **Idea**: For \( A(x,y)/B(y) \), compute all \( x \) values that are not 'disqualified' by some \( y \) value in \( B \).
  - \( x \) value is disqualified if by attaching \( y \) value from \( B \), we obtain an \( xy \) tuple that is not in \( A \).

  Disqualified \( x \) values: \[
  \pi_x \left( \pi_y (A(x,y) \times B) \setminus A \right)
  \]

  \( A/B \): \[
  \pi_x (A) \setminus \text{ Disqualified } x \text{ values}
  \]

**Examples**

Find names of sailors who've reserved boat #103

• **Solution 1**: \[
\pi_{\text{sname}} (\sigma_{\text{bid}=103} \text{Reserves} \bowtie_{\text{sid}=\text{bid}} \text{Sailors})
\]

• **Solution 2**: \[
\pi_{\text{sname}} (\sigma_{\text{bid}=103} \text{Reserves} \bowtie_{\text{sid}=\text{bid}} \text{Sailors})
\]
Find names of sailors who’ve reserved a red boat
- Information about boat color only available in Boats; so need an extra join:
  $$\pi_{\text{name}}(\sigma_{\text{color} = 'red'} \text{Boats} \bowtie \text{Reserves} \bowtie \text{Sailors})$$
- A more efficient solution:
  $$\pi_{\text{name}}(\sigma_{\text{sid}}(\pi_{\text{bid}}(\sigma_{\text{color} = 'red'} \text{Boats} \bowtie \text{Reserves} \bowtie \text{Sailors})))$$

- A query optimizer can find this given the first solution!

Find sailors who’ve reserved a red or a green boat
- Can identify all red or green boats, then find sailors who’ve reserved one of these boats:
  $$\rho((\text{Tempboats}, (\sigma_{\text{color} = 'red'} \lor \text{color} = 'green') \text{Boats})), \pi_{\text{name}}(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors})$$

Find sailors who’ve reserved a red and a green boat
- Previous approach won’t work! Must identify sailors who’ve reserved red boats, sailors who’ve reserved green boats, then find the intersection (note that sid is a key for Sailors):
  $$\rho((\text{Tempred}, \pi_{\text{sid}}((\sigma_{\text{color} = 'red'} \text{Boats} \bowtie \text{Reserves} \bowtie \text{Sailors}))), \rho((\text{Tempgreen}, \pi_{\text{sid}}((\sigma_{\text{color} = 'green'} \text{Boats} \bowtie \text{Reserves}))), \pi_{\text{name}}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors}))$$

Find sailors who’ve reserved all boats
- Uses division; schemas of the input relations to / must be carefully chosen:
  $$\rho((\text{Tempsids}, (\pi_{\text{sid, bid}} \text{Reserves})), (\pi_{\text{bid}} \text{Boats})))$$
  $$\pi_{\text{name}}(\text{Tempsids} \bowtie \text{Sailors})$$
- To find sailors who’ve reserved all ‘Interlake’ boats:
  $$\ldots / \pi_{\text{bid}}((\sigma_{\text{bname} = 'Interlake'} \text{Boats}))$$

Summary
- Relational Algebra: a small set of operators mapping relations to relations
  - Operational, in the sense that you specify the explicit order of operations
  - A closed set of operators! Can mix and match.
- Basic ops include: $\sigma_a, \pi_x, \cup, -$;
- Important compound ops: $\cap$, $\times$, $/$