Relational Algebra

R & G, Chapter 4

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!

Relational Algebra: 5 Basic Operations

- **Selection** () Selects a subset of rows from relation (horizontal).
- **Projection** () Retains only wanted columns from relation (vertical).
- **Cross-product** (X) Allows us to combine two relations.
- **Set-difference** (–) Tuples in r1, but not in r2.
- **Union** (∪) Tuples in r1 and/or in r2.

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

Example Instances

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

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

Boats

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
Projection

- Examples: $\cup_{\text{age}(S2)}$, $\cup_{\text{name, rating}(S2)}$
- Retains only attributes that are in the "projection list".

- **Schema** of result:
  - exactly the fields in the projection list, with the same names that 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?

\[
\begin{array}{cccc}
\text{sid} & \text{name} & \text{rating} & \text{age} \\
28 & \text{yuppy} & 9 & 35.0 \\
31 & \text{lubber} & 8 & 55.5 \\
44 & \text{guppy} & 5 & 35.0 \\
58 & \text{rusty} & 10 & 35.0 \\
\end{array}
\]

Selection ($\mathcal{S}$)

- 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?

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

Union

- All 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?

\[
\begin{array}{cccc}
\text{sid} & \text{name} & \text{rating} & \text{age} \\
22 & \text{dustin} & 7 & 45.0 \\
31 & \text{lubber} & 8 & 55.5 \\
44 & \text{guppy} & 5 & 35.0 \\
58 & \text{rusty} & 10 & 35.0 \\
\end{array}
\]

Set Difference

\[
\begin{array}{cccc}
\text{sid} & \text{name} & \text{rating} & \text{age} \\
28 & \text{yuppy} & 9 & 35.0 \\
31 & \text{lubber} & 8 & 55.5 \\
44 & \text{guppy} & 5 & 35.0 \\
58 & \text{rusty} & 10 & 35.0 \\
\end{array}
\]
Cross-Product

• S1 \(\times\) R1: Each row of S1 paired with each row of R1.
• Q: How many rows in the result?
• Result schema has one field per field of S1 and R1, with field names “inherited” if possible.
  – May have a naming conflict: Both S1 and R1 have a field with the same name.
  – In this case, can use the renaming operator: \(\square(C(1 \times \text{sid}1, 5 \times \text{sid}2), S1 \times R1)\)

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>

R1 \(\times\) S1 =

<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>

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>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>

S1 \(\times\) S2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<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>

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>

R1 \(\bowtie\) S1 =

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</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?
  \(R \cap S = R \setminus (R \setminus S)\)

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.
• Useful for putting “normalized” relations back together.
Other Types of Joins

- **Condition Join (or "theta-join"):**
  \[ R \bowtie_c S = \bigcap_c (R \setminus S) \]

  - **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.

  \[ R \bowtie_s \bowtie_t ... \bowtie_k S \]

  - May have fewer tuples than cross-product.

Examples of Division A/B

<table>
<thead>
<tr>
<th>sno</th>
<th>pno</th>
<th>pno</th>
<th>pno</th>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td>p2</td>
<td>p2</td>
<td>p2</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
<td>p1</td>
<td>p3</td>
<td>p4</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
<td>s1</td>
<td>p4</td>
<td>s4</td>
</tr>
<tr>
<td>s2</td>
<td>s1</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
<td>s2</td>
<td>s4</td>
<td>s1</td>
</tr>
<tr>
<td>s3</td>
<td>s4</td>
<td>s2</td>
<td>s3</td>
<td>s4</td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
<td>s4</td>
<td>s1</td>
<td>s4</td>
</tr>
</tbody>
</table>

\( A \bowtie B1 \bowtie B2 \bowtie B3 \)

Examples

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

<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>

Compound Operator: Division

- Useful for expressing “for all” queries like: Find units of sailors who have reserved all boats.
- For A/B attributes of B are subset of 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 \( y \):

  \[ A/B = \{ (x) | \forall y (B(x, y) \land A) \} \]

  \( A/B \) contains all tuples \( (x) \) such that for every \( y \) tuple in \( B \), there is an \( xy \) 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/B, 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.

  \[ \{ x \} \setminus \bigcap_{y} B(x, y) \bigcap A \]

  \( A/B \) contains all tuples \( (x) \) such that for every \( y \) tuple in \( B \), there is an \( xy \) tuple in \( A \).

Find names of sailors who’ve reserved boat #103

- **Solution 1:** \( \exists_{\text{name}} \left( \bowtie_{\text{bid}=103} \text{Reserves} \right) \bowtie \text{Sailors} \)

- **Solution 2:** \( \exists_{\text{name}} \left( \bowtie_{\text{bid}=103} \left( \text{Reserves} \right) \right) \bowtie \text{Sailors} \)
Find names of sailors who’ve reserved a red boat

- Information about boat color only available in Boats; so need an extra join:
  \[ \pi_{\text{color}}(\pi_{\text{red}}(\text{Boats}) \bowtie \text{Reserves}) \bowtie \text{Sailors} \]

  A more efficient solution:
  \[ \pi_{\text{color}}(\pi_{\text{red}}(\text{Boats})) \bowtie \text{Reserves} \bowtie \text{Sailors} \]

  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:
  \[ \pi_{\text{color}}(\pi_{\text{red}} \cup \pi_{\text{green}}(\text{Boats})) \bowtie \text{Reserves} \bowtie \text{Sailors} \]

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):
  \[ \pi_{\text{sid}}(\pi_{\text{red}}(\text{Boats}) \bowtie \text{Reserves}) \bowtie \text{Sailors} \]
  \[ \pi_{\text{sid}}(\pi_{\text{green}}(\text{Boats}) \bowtie \text{Reserves}) \bowtie \text{Sailors} \]
  \[ \pi_{\text{name}}(\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors} \]

Find the names of sailors who’ve reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:
  \[ \pi_{\text{bid}}(\pi_{\text{sid}}(\text{Reserves}) / (\pi_{\text{bid}}(\text{Boats})) \bowtie \text{Sailors} \]

  To find sailors who’ve reserved all ’Interlake’ boats:
  \[ \pi_{\text{bid}}(\pi_{\text{name}}=\text{Interlake}(\text{Boats})) \bowtie \text{Sailors} \]

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: \( \bowtie, \cup, \cap, - \)
- Important compound ops: \( \bowtie, \cup, \cap, / \)