Relational Algebra

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 (procedural), very useful for representing execution plans.
  - **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-operational, declarative.)

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 regardless of instance!)
  - The schema for the result of a given query is also fixed! Determined by definition of query language constructs.
- **Positional vs. named-field notation**:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL
Example Instances

<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>uppy</td>
<td>9</td>
<td>35.0</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>

"Sailors" and "Reserves" relations for our examples.

We'll use positional or named field notation, assume that names of fields in query results are 'inherited' from names of fields in query input relations.

Projection

- Deletes attributes that are not in projection list.
- Schema of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate duplicates! (Why??, what are the consequences?)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

Selection

- Selects rows that satisfy selection condition.
- Schema of result identical to schema of (only) input relation.
- Result relation can be the input for another relational algebra operation! (Operator composition.)

Relational Algebra

- Basic operations:
  - Selection (\(\sigma\)): Selects a subset of rows from relation.
  - Projection (\(\pi\)): Deletes unwanted columns from relation.
  - Cross-product (\(\times\)): Allows us to combine two relations.
  - Set-difference (\(\ominus\)): Tuples in reln. 1, but not in reln. 2.
  - Union (\(\cup\)): Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, join, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be composed! (Algebra is "closed".)
### Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be union-compatible:
  - Same number of fields.
  - 'Corresponding' fields have the same type.
- What is the schema of result?

<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>rubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
</tbody>
</table>

### Joins

- **Condition Join:** $R \bowtie_c S = \sigma_c (R \times S)$

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>rubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- **Equi-Join:** $S \bowtie_{\text{sid}=\text{bid}} R$
- **Natural Join:** Equijoin on all common fields.

### Cross-Product

- Each row of $S_1$ is paired with each row of $R_1$.
- Result schema has one field per field of $S_1$ and $R_1$, with field names 'inherited' if possible.
- **Conflict:** Both $S_1$ and $R_1$ have a field called sid.

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>rubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- **Renaming operator:** $\rho (C(1 \rightarrow \text{sid}, 5 \rightarrow \text{sid}2), S_1 \times R_1)$

---

The text above describes various database operations, including union, intersection, set-difference, and joins. It explains the conditions and results of these operations, providing examples and clear explanations.
Division

- Not supported as a primitive operator, but useful for expressing queries like:
  - Find sailors who have reserved **all** boats.
- Precondition: in A / B, the attributes in B must be included in the schema for A. Also, the result has attributes A-B.
  - `SALES(supId, prodId)`;
  - `PRODUCTS(prodId)`;
  - Relations `SALES` and `PRODUCTS` must be built using projections.
  - `SALES/PRODUCTS`: the ids of the suppliers supplying **all** products.

Examples of Division A/B

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

A | A/B1 | A/B2 | A/B3

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. Division is NOT implemented in SQL).
- Idea: For `SALES/PRODUCTS`, compute all products such that there exists at least one supplier not supplying it.
  - `x` value is disqualified if by attaching `y` value from B, we obtain an `xy` tuple that is not in A.

\[
A = \pi_{sid}(\pi_{bid}(Sales \times Products) - Sales)
\]

The answer is \(\pi_{sid}(Sales) - A\)

Find names of sailors who’ve reserved boat #103

- Solution 1: \(\pi_{sname}(\sigma_{bid=103}(Reserves)) \bowtie_{\bowtie} Sailors\)
- Solution 2: \(\rho(Temp_1, \sigma_{bid=103}(Reserves)) \bowtie_{\bowtie} Sailors\)
  \(\rho(Temp_2, Temp_1 \bowtie_{\bowtie} Sailors)\)
  \(\pi_{sname}(Temp_2)\)
- Solution 3: \(\pi_{sname}(\sigma_{bid=103}(Reserves) \bowtie_{\bowtie} Sailors)\)
Find names of sailors who've reserved a red boat

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

- A more efficient solution:
  \[ \pi_{\text{name}}(\pi_{\text{sid}}(\sigma_{\text{bid}}(\sigma_{\text{color}=\text{red}}(\text{Boats}))) \bowtie_{\text{Reserves}} \bowtie_{\text{Sailors}}) \]

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

Find sailors who've reserved a red 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):
  \[ \rho(\text{Tempred}, \pi_{\text{sid}}(\sigma_{\text{color}=\text{red}}(\text{Boats}))) \bowtie_{\text{Reserves}} \]

  \[ \rho(\text{Tempgreen}, \pi_{\text{sid}}(\sigma_{\text{color}=\text{green}}(\text{Boats}))) \bowtie_{\text{Reserves}} \]

  \[ \pi_{\text{name}}(\text{Tempred} \land \text{Tempgreen}) \bowtie_{\text{Sailors}} \]

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:
  \[ \rho(\text{Tempboats}, (\sigma_{\text{color}=\text{red}} \lor \text{color}=\text{green})(\text{Boats})) \]

  \[ \pi_{\text{name}}(\text{Tempboats} \bowtie_{\text{Reserves}} \bowtie_{\text{Sailors}}) \]

- Can also define Tempboats using union! (How?)

- What happens if \( \lor \) is replaced by \( \land \) in this query?

Find the names of sailors who've reserved all boats

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

  \[ \pi_{\text{name}}(\text{Tempids} \bowtie_{\text{Sailors}}) \]

- To find sailors who've reserved all ‘Interlake’ boats:
  \[ \ldots \land \pi_{\text{bid}}(\sigma_{\text{bname}=\text{Interlake}}(\text{Boats})) \]
Summary

- The relational model has rigorously defined query languages that are simple and powerful.
- Relational algebra is more operational; useful as internal representation for query evaluation plans.
- Several ways of expressing a given query; a query optimizer should choose the most efficient version.