Relational Algebra

Chapter 4, Part A

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-operational, 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 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

- “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.

<table>
<thead>
<tr>
<th>R1</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S1</th>
<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>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S2</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

Relational Algebra

- **Basic operations**:
  - **Selection (σ)** Selects a subset of rows from relation.
  - **Projection (Pi)** Deletes unwanted columns from relation.
  - **Cross-product (X)** Allows us to combine two relations.
  - **Set-difference (−)** Tuples in reln. 1, but not in reln. 2.
  - **Union (U)** Tuples in reln. 1 and in reln. 2.

- **Additional operations**:
  - Intersection, **adjacency**, division, renaming: Not essential, but (very!) useful.
  - Since each operation returns a relation, operations can be composed! (Algebra is “closed”.)
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?)
  - 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.
- No duplicates in result! (Why?)
- Schema of result identical to schema of (only) input relation.
- Result relation can be the input for another relational algebra operation. (Operator composition.)

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?

Cross-Product

- Each row of S1 is paired with each row of R1.
- Result schema has one field per field of S1 and R1, with field names 'inherited' if possible.
  - Conflict: Both S1 and R1 have a field called sid.

Joins

- Condition Join: \( R[U(c)] S = \sigma_c (R \times S) \)
- Equi-Join: A special case of condition join where the condition \( c \) contains only equalities.
- Equi-join: Equijoin on all common fields.

Natural Join: Equijoin on all common fields.
Division

- Not supported as a primitive operator, but useful for expressing queries like:
  Find sailors who have reserved all boats.
- Let A have 2 fields, x and y; B have only field y:
  - \( A/B = \{ (x) \mid \exists (x, y) \in A \forall (y) \in B \} \)
  - i.e., \( A/B \) contains all x tuples (sailors) such that for every y tuple (boat) in B, there is an xy tuple in A.
- Or: If the set of y values (boats) associated with an x value (sailor) in A contains all y values in B, the x value is in A/B.
- In general, x and y can be any lists of fields; y is the list of fields in B, and xy is the list of fields of A.

Examples of Division A/B

\[
\begin{array}{c|c|c|c}
\text{sno} & \text{pno} & \text{pno} & \text{pno} \\
\hline
s1 & p1 & B1 & p1 \\
s1 & p2 & B2 & p2 \\
s1 & p3 & & p4 \\
s1 & p4 & & p4 \\
s2 & p1 & sno & \\
s2 & p2 & & sno \\
s3 & p2 & & sno \\
s4 & p2 & & sno \\
s4 & p4 & & sno \\
\end{array}
\]

\( A/ \{A/B1, A/B2, A/B3\} \)

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.)
- 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.
  - Disqualified x values: \( \pi_x (\pi_x (A) \times B) - A) \)
  - \( A/B = \pi_x (A) - \text{all disqualified tuples} \)

Find names of sailors who’ve reserved boat #103

- Solution 1: \( \pi_{\text{sname}} (\sigma_{\text{bid}=103} (\text{Reserves}) \bowtie \text{Sailors}) \)
- Solution 2: \( \rho (\text{Temp1}, \sigma_{\text{bid}=103} (\text{Reserves}) \)
  \( \rho (\text{Temp2}, \text{Temp1} \bowtie \text{Sailors}) \)
  \( \pi_{\text{sname}} (\text{Temp2}) \)
- Solution 3: \( \pi_{\text{sname}} (\sigma_{\text{bid}=103} (\text{Reserves} \bowtie \text{Sailors})) \)

Find names of sailors who’ve reserved a red boat

- Information about boat color only available in Boats; so need an extra join:
  \( \pi_{\text{sname}} (\sigma_{\text{color} = \text{'red'}} (\text{Boats}) \bowtie \text{Reserves} \bowtie \text{Sailors}) \)
- A more efficient solution:
  \( \pi_{\text{sname}} (\sigma_{\text{sid}} ((\sigma_{\text{bid}} \sigma_{\text{color} = \text{'red'}} (\text{Boats}) \bowtie \text{Res}) \bowtie \text{Sailors})) \)
- 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:
  \( \rho (\text{Tempboats}, (\sigma_{\text{color} = \text{'red'}} \lor \text{color} = \text{'green'}} (\text{Boats})) \)
  \( \pi_{\text{sname}} (\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors}) \)
- Can also define Tempboats using union! (How?)
- What happens if \( \lor \) is replaced by \( \land \) in this query?
Find 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):

\[
\rho (\text{Tempred}, \pi_{\text{sid}}((\sigma_{\text{color}=\text{red}} \text{Boats}) \bowtie \text{Reserves})) \\
\rho (\text{Tempgreen}, \pi_{\text{sid}}((\sigma_{\text{color}=\text{green}} \text{Boats}) \bowtie \text{Reserves})) \\
\pi_{\text{sname}}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors})
\]

Find the names of sailors who’ve reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:

\[
\rho (\text{Tempsids}, (\pi_{\text{sid,bid}} \text{Reserves}) / (\pi_{\text{bid}} \text{Boats})) \\
\pi_{\text{sname}}(\text{Tempsids} \bowtie \text{Sailors})
\]

To find sailors who’ve reserved all ‘Interlake’ boats:

\[
\cdots / \pi_{\text{bid}}((\sigma_{\text{bname}=\text{Interlake}} \text{Boats})
\]

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.