Introduction to Data Management

Lecture #11
(Relational Algebra)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- HW and exams:
  - HW #3 info
    - Due today (or next 2 days, if late)
    - Solution will be published on Sunday
  - Midterm Exam 1 on Monday (at this time)!
    - You may create and use an 8.5”x11” “cheat sheet”
    - You must sit in your assigned seat (check seating chart)
    - No discussion sessions next week!

- Today’s plan:
  - Relational query languages, Episode 1
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 their 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.)
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 (but try to avoid positional stuff!)

Example Instances

- “Sailors” and “Reserves” relations for our examples.
- We’ll use positional or named field notation, and assume that names of fields in query results are “inherited” from names of fields in query input relations (when possible).

<table>
<thead>
<tr>
<th></th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</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>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>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>
Relational Algebra

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

- Additional operations:
  - Intersection, join, division, renaming: Not essential, but (very!) useful. (I.e., don’t add expressive power, but…)

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

Projection

- Removes 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.
- Relational projection operator has to eliminate duplicates! (Why??)
  - Note: real systems typically don’t do duplicate elimination unless the user explicitly asks for it. (Q: Why not?)

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>yacht</td>
<td>9</td>
</tr>
<tr>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>guppy</td>
<td>5</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

\[ \pi_{sname, rating}(S2) \]

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.0</td>
</tr>
<tr>
<td>55.5</td>
</tr>
</tbody>
</table>

\[ \pi_{age}(S2) \]
Selection

- Selects rows that satisfy a selection condition.
- No duplicates in result! (Why?)
- Schema of result identical to schema of its (only) input relation.
- Result relation can be the input for another relational algebra operation! (This is operator composition.)

\[ \sigma_{\text{rating} > 8}(S) \]

\[
\begin{array}{l|l|l|l}
\text{sid} & \text{sname} & \text{rating} & \text{age} \\
28 & \text{yuppy} & 9 & 35.0 \\
58 & \text{rusty} & 10 & 35.0 \\
\end{array}
\]

\[
\pi_{\text{sname}, \text{rating}}(\sigma_{\text{rating} > 8}(S))
\]

Union, Intersection, Set-Difference

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

\[ S_1 \cup S_2 \]

\[
\begin{array}{l|l|l|l|l}
\text{sid} & \text{sname} & \text{rating} & \text{age} \\
22 & \text{dustin} & 7 & 45.0 \\
31 & \text{lubber} & 8 & 55.5 \\
58 & \text{rusty} & 10 & 35.0 \\
44 & \text{guppy} & 5 & 35.0 \\
28 & \text{yuppy} & 9 & 35.0 \\
\end{array}
\]

\[ S_1 \cap S_2 \]

\[
\begin{array}{l|l|l|l|l}
\text{sid} & \text{sname} & \text{rating} & \text{age} \\
31 & \text{lubber} & 8 & 55.5 \\
58 & \text{rusty} & 10 & 35.0 \\
\end{array}
\]

\[ S_1 - S_2 \]

Q: Any issues w/duplicates?
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.

<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>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<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>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- *Renaming operator*: $\rho (C(l \rightarrow \text{sid}1, 5 \rightarrow \text{sid}2), S1 \times R1)$

Renaming

- *Conflict*: S1 and R1 both had sid fields, giving:

<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>22</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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- Several renaming options available:

  - Positional renaming
  - Name-based renaming
  - Generalized projection

  $\rho (S1R1(l \rightarrow \text{sid}1), S1 \times R1)$

  $\rho (\text{TempS}1(\text{sid} \rightarrow \text{sid}1), S1)$

  TempS1$\times$R1

  $(\pi_{\text{sid} \rightarrow \text{sid}1, \text{sname}, \text{rating}, \text{age}}(S1)) \times R1$
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>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

\( S_1 \bowtie_{sid=sid} R_1 \)

- **Result schema** same as that of cross-product.
- Fewer tuples than cross-product, so might be able to compute more efficiently.
- Sometimes (often!) called a **theta-join**.

More Joins

- **Equi-Join**: A special case of condition join where the condition \( c \) contains only **equalities**.

<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>

\( S_1 \bowtie_{sid} R_1 \)

- **Result schema** similar to cross-product, but only one copy of fields for which equality is specified.
- **Natural Join**: An equijoin on **all** commonly named fields.
Division

- Not a primitive operator, but extremely useful for expressing queries like:
  
  *Find sailors who have reserved all boats.*

- Let $A$ have 2 fields, $x$ and $y$, while $B$ has one field $y$, so we have relations $A(x,y)$ and $B(y)$:
  
  - $A/B$ contains the $x$ tuples (e.g., sailors) such that for every $y$ tuple (e.g., 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 $x \cup y$ is the list of fields of $A$.

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

$A$ $A/B1$ $A/B2$ $A/B3$
Expressing A/B Using Basic Operators
(Advanced Topic 🎓)

- Division not an essential op; just a useful shorthand.
  (Also true of joins, but joins are so common and important that relational database 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 a y value from B, we obtain an xy tuple that does not appear in A.

\[ D(x) = \pi_x ((\pi_x (A) \times B) - A) \]
A/B: \[ \pi_x (A) - D \]

To be continued...