Introduction to Data Management

Lecture #11
(Relational Languages I)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- Homework stuff
  - HW #3 is due today (tomorrow if late)
  - HW #4 will come out on Friday (after the exam)
- Exam stuff (time flies!)
  - Midterm #1 is this Friday (in class) – i.e., in two days!
  - We’ll use assigned seating – come early!
  - You may bring an 8.5”x11” (2-sided) cheat sheet (as long as it has your name and seating location on it!)
- Today’s plan:
  - Relational languages – the next frontier
  - BTW: Today’s material isn’t on Friday’s exam!… 😊
But 1st: A Word on Learning….

- "When *I* was your age…” (😉)
  - 8-bit μprocessors, LEDs, hex keypad, 16-bit for its address space, …
  - Unix appeared, Unix/INGRES DB 64K process design point, then 1MB of memory was amazing!
  - FORTRAN, Pascal, C, Lisp, Snobol, APL, Quel …
  - No web! (Just the early ArpaNet)

- Fast forward 35 years…
  - Your cell phones dwarf our ~1980 computing platforms
  - Python, Java, Go, C++, Ruby, SQL, SQL++, …
  - Twitter… (😉)
  - WHAT THIS MEANS: Critical to learn how to read and learn, how to search for info/resources, etc…!!

Reminder: Normal Forms

- All “relations”
  - 1NF
  - 2NF
  - 3NF
  - BCNF
  - …
Relational Design Theory Summary

- If a relation is in **BCNF**, it is free of redundancies that can be detected using FDs. (Trying to ensure that all relations are in BCNF is thus a good goal.)
- If a relation is not in BCNF, we can decompose it into a lossless-join collection of BCNF relations.
  - Are all FDs preserved? If a lossless-join, dependency-preserving decomposition into BCNF is not possible (or is unsuitable for typical queries), consider **3NF** instead.
  - Note: Decompositions should be carried out while also keeping *performance requirements* in mind. (More later!)

Note: Refining an ER Based Design

- 1st diagram translated:
  - Workers($S,N,L,D,S$)
  - Departments($D,M,B$)
    - Lots associated with workers.
  - **Suppose all workers in a dept are assigned the same lot:** $D \rightarrow L$ ....
- Redundancy; fixed by:
  - Workers2($S,N,D,S$)
  - WorkersLots($D,L$)
  - Departments($D,M,B$)
- Can further fine-tune this:
  - Workers2($S,N,D,S$)
  - Departments($D,M,B,L$)

Notice: Lot wasn’t really a “Worker attribute”!

Before:

After:
### PS: On Refining ER Based Designs

- 1st diagram translated:
  - Workers(S, N, L, D, S)
  - Departments(D, M, B)

  **Suppose all workers in a dept are assigned the same lot:**
  - D \rightarrow L ....

- Redundancy; fixed by:
  - Workers2(S, N, D, S)
  - Departments(D, M, B, L)

  **Can further fine-tune this:**
  - Workers2(S, N, D, S)
  - Departments(D, M, B, L)

**Note:**
- In many cases the relational translation of an ER design will take you right to 3NF (and BCNF)…!
  - Entity key \rightarrow attributes for entity sets.
  - Relationship key \rightarrow attributes for relationship sets.

  (But problems can exist with FDs between attributes.)

---

### Lingering Questions…?

---
On to 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>surname</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>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>puppy</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** \((\sigma)\) Selects a subset of rows from relation.
  - **Projection** \((\pi)\) Omits unwanted columns from relation.
  - **Cross-product** \((\times)\) Allows us to combine two relations.
  - **Set-difference** \((-\phantom{1})\) Tuples in reln. 1, but not in reln. 2.
  - **Union** \((\cup\phantom{1})\) 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?)

\[
\pi \text{sname, rating (S2)} \\
\begin{array}{|c|c|}
\hline
\text{sname} & \text{rating} \\
\hline
\text{yuppy} & 9 \\
\text{lubber} & 8 \\
\text{guppy} & 5 \\
\text{rusty} & 10 \\
\hline
\end{array}
\]
### 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.)

### 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?

**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.

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

**Renaming operator**: \( \rho (C(l \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1) \)

Renaming

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>sname</td>
<td>rating</td>
<td>age</td>
<td>bid</td>
<td>day</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
</tr>
</tbody>
</table>

- Several renaming options available:

\[ \rho (S1R1(l \rightarrow sid1), S1 \times R1) \]
\[ \rho (TempS1(sid \rightarrow sid1), S1) \]
\[ TempS1 \times R1 \]
\[ (\pi _{sid \rightarrow sid1,sname,rating,age} (S1)) \times R1 \]
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>lumber</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*. 

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke