CompSci 516
Database Systems
(Incomplete Notes)

Lecture 4
Relational Algebra
and
Relational Calculus

Instructor: Sudeepa Roy

Announcements

• Reminder: HW1
  – Sakai: Resources -> HW -> HW1 folder
  – Due on 09/20 (Thurs), 11:55 pm, no late days
  – Start now!
  – Submission instructions for gradescope to be updated (will be notified through piazza)
• Your piazza and sakai accounts should be active
  – if not on piazza, send me an email

Recap: SQL -- Lecture 2/3

• Creating/modifying relations
• Specifying integrity constraints
• Key/candidate key, superkey, primary key, foreign key
• Conceptual evaluation of SQL queries
• Joins
• Group bys and aggregates
• Nested queries
• NULLs
• Views

Today’s topics

• Relational Algebra (RA) and Relational Calculus (RC)
• Reading material
  – [RG] Chapter 4 (RA, RC)
  – [GUW] Chapters 2.4, 5.1, 5.2

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

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 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, or procedural)
- Note: Declarative (RC, SQL) vs. Operational (RA)

Preliminaries (recap)

- 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
    - 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

Example Schema and Instances

<table>
<thead>
<tr>
<th>Sailors (sid, sname, rating, age)</th>
<th>Boats (bid, bname, color)</th>
<th>Reserves (sid, bid, day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>sname</td>
<td>rating</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</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>yuppy</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

- Takes one or more relations as input, and produces a relation as output
  - operator
  - operand
  - semantic
  - so an algebra!
- Since each operation returns a relation, operations can be composed
  - Algebra is “closed”
Relational Algebra

- **Basic operations:**
  - Selection (σ) Selects a subset of rows from relation
  - Projection (π) 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 (∪) Tuples in reln. 1 or in reln. 2.

- **Additional operations:**
  - Intersection (∩)
  - Join (⋈)
  - Division (÷)
  - Renaming (ρ)
  - Not essential, but very useful.

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.

Composition of Operators

- Result relation can be the input for another relational algebra operation

- Operator composition

  \[ \sigma_{\text{rating} > 8}(S2) \]

  \[ \pi_{\text{name, rating}}(\sigma_{\text{rating} > 8}(S2)) \]

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.
  - Same schema as the inputs.

**Note:** no duplicate

- "Set semantic"
- SQL: UNION
- SQL allows "bag semantic" as well: UNIVI ALL
Union, Intersection, Set-Difference

<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>
<tr>
<td>S2</td>
<td>sid</td>
<td>sname</td>
<td>rating</td>
<td>age</td>
</tr>
<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>

S1 − S2

S1 ∩ S2

Cross-Product

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

<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>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>

Renaming Operator \( \rho \)

\( \rho_{\text{sid} \rightarrow \text{sid}1} \, S1 \) \( \times \) \( \rho_{\text{sid} \rightarrow \text{sid}1} \, R1 \)

or

\( \rho(\text{C}(1 \rightarrow \text{sid}1, 5 \rightarrow \text{sid}2), \, S1 \times R1) \)

C is the new relation name

- In general, can use \( \rho(<\text{Temp}> , <\text{RA-expression}>) \)

Find names of sailors who’ve reserved boat #103

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

Joins

\( R \bowtie_c S = \sigma_c (R \times S) \)

\( S1 \bowtie_{S1.\text{sid} < R1.\text{sid}} R1 \)

- Result schema same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently

Find names of sailors who’ve reserved boat #103

Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

- Solution 1:
- Solution 2:
Expressing an RA expression as a Tree

Find sailors who’ve reserved a red or a green boat

What about aggregates?

Relational Calculus (RC)

Relational Calculus

TRC: example
A ⇒ B

• A “implies” B
• Equivalently, if A is true, B must be true
• Equivalently, ¬ A ∨ B, i.e.
  — either A is false (then B can be anything)
  — otherwise (i.e. A is true) B must be true

Useful Logical Equivalences

• ∃x P(x) = ¬¬∃x [¬P(x)]
• ¬(P ∨ Q) = ¬ P ∧ ¬ Q, de Morgan’s laws
  Similarly, ¬(¬P ∨ Q) = P ∧ ¬ Q, etc.

• A ⇒ B = ¬ A ∨ B

TRC: example
Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

• Find the names of sailors who have reserved at least two boats

TRC: example
Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

• Find the names of sailors who have reserved at least two boats

TRC: example
Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

• Find the names of sailors who have reserved all boats
• Called the “Division” operation

TRC: example
Sailors(sid, sname, rating, age)
Boats(bid, bname, color)
Reserves(sid, bid, day)

• Find the names of sailors who have reserved all boats
• Division operation in RA!
• Find the names of sailors who have reserved all red boats

Recall that \( A \Rightarrow B \) is logically equivalent to \( \neg A \lor B \) so \( \Rightarrow \) can be avoided, but it is cleaner and more intuitive.

• Variables are now domain variables
• We will use TRC.
  — both are equivalent
• Another option to write coming soon!

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y, z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

A shortcut for:

\[ \{ x \mid \forall y \in \text{Frequents} \land \exists z \in \text{Serves} \land \exists w \in \text{Likes} \} \]

The difference is that in the first one, one variable = one attribute in the second one, one variable = one tuple (Tuple RC)
Both are equivalent and feel free to use the one that is convenient to you.
Drinker Category 2

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent only bars that serves some beer they like.

\[ Q(x) = \ldots \]

Drinker Category 3

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent only bars that serves some beer they like.

Find drinkers that frequent some bar that serves only beers they like.

Drinker Category 4

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent only bars that serves some beer they like.

Find drinkers that frequent some bar that serves only beers they like.

Find drinkers that frequent only bars that serves only beer they like.
Why should we care about RC

- RC is declarative, like SQL, and unlike RA (which is operational)
- Gives foundation of database queries in first-order logic
  - you cannot express all aggregates in RC, e.g. cardinality of a relation or sum (possible in extended RA and SQL)
  - still can express conditions like “at least two tuples” (or any constant)
- RC expression may be much simpler than SQL queries
  - and easier to check for correctness than SQL
  - power to use \( \land \) and \( \Rightarrow \)
    - then you can systematically go to a “correct” SQL query

From RC to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

\[
Q(x) = \exists y. \text{Likes}(x, y) \land \forall z. (\text{Serves}(z, y) \Rightarrow \text{Frequents}(x, z))
\]

Step 1: Replace \( \lor \) with \( \land \) using de Morgan’s Laws

\[
Q(x) = \exists y. \text{Likes}(x, y) \land \forall z. (\neg \text{Serves}(z, y) \lor \neg \text{Frequents}(x, z))
\]

From RC to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

\[
Q(x) = \exists y. \text{Likes}(x, y) \land \forall z. (\neg \text{Serves}(z, y) \lor \neg \text{Frequents}(x, z))
\]

Step 2: Translate into SQL

```
SELECT DISTINCT L.drinker
FROM Likes L
WHERE not exists
(SELECT S.bar
FROM Serves S
WHERE L.beer = S.beer
AND not exists (SELECT *
FROM Frequents F
WHERE F.drinker = L.drinker
AND F.bar = S.bar))
```

We will see a “methodical and correct” translation through “safe queries” in Datalog

Summary

- You learnt three query languages for the Relational DB model
  - SQL
  - RA
  - RC
- All have their own purposes
- You should be able to write a query in all three languages and convert from one to another
  - However, you have to be careful, not all “valid” expressions in one may be expressed in another
  - \( \{ s | \neg \{ s : \text{Sailors} \} \} \) – infinitely many tuples – an “unsafe” query
  - More when we do “Datalog”, also see Ch. 4.4 in [RG]