CompSci 516
Database Systems

Lecture 4
Relational Algebra
and
Relational Calculus

Instructor: Sudeepa Roy

Announcements
• In-person classes starting Thursday
  – Also live streaming and recording

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

Relational Query Languages

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

Logic Notations

- $\exists$ There exists
- $\forall$ For all
- $\wedge$ Logical AND
- $\vee$ Logical OR
- $\neg$ NOT
- $\Rightarrow$ Implies

Relational Algebra (RA)

- 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 (-) Tuples in reln. 1, but not in reln. 2.
  - Union (U) Tuples in reln. 1 or in reln. 2.
- Additional operations:
  - Intersection ($\cap$)
  - join $\land$
  - division (/)
  - renaming ($\rho$)
  - Not essential, but (very) useful.
### Projection

- Deletes attributes that are not in the 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 (performance)

\[ \pi_{\text{name}, \text{rating}}(S^2) \]

### Selection

- Selects rows that satisfy selection condition
- No duplicates in result. Why?
- Schema of result identical to schema of (only) input relation

\[ \sigma_{\text{rating} > 8}(S^2) \]

### Composition of Operators

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

\[ \pi_{\text{name}, \text{rating}}(\sigma_{\text{rating} > 8}(S^2)) \]

### 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

\[ S^1 \cup S^2 \]

\[ S^1 \cap S^2 \]

\[ S^1 \setminus S^2 \]

\[ S^1 \cap S^2 \]

### Note:

- No duplicate
- "Set semantic"
- SQL: UNION
- SQL allows "bag semantic" as well: UNION ALL

\[ \begin{array}{|c|c|c|c|}
\hline
\text{sid} & \text{name} & \text{rating} & \text{age} \\
\hline
28 & puppy & 9 & 35.0 \\
31 & lubber & 8 & 55.5 \\
44 & guppy & 5 & 35.0 \\
58 & rusty & 10 & 35.0 \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|}
\hline
\text{sid} & \text{name} & \text{rating} & \text{age} \\
\hline
22 & dustin & 7 & 45.0 \\
31 & lubber & 8 & 55.5 \\
58 & rusty & 10 & 35.0 \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|}
\hline
\text{sid} & \text{name} & \text{rating} & \text{age} \\
\hline
28 & puppy & 9 & 35.0 \\
31 & lubber & 8 & 55.5 \\
44 & guppy & 5 & 35.0 \\
58 & rusty & 10 & 35.0 \\
\hline
\end{array} \]
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 ρ

(ρ_{sid → sid1} S1) × (ρ_{sid → sid1} R1)
or
ρ(C(1→ sid1, 5 → sid2), S1 × R1)

In general, can use ρ(Temp, RA-expression)

Find names of sailors who’ve reserved boat #103

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

• Solution 1: \[ \pi_{sname}(\sigma_{bid=103}(\text{Reserves}) \Join \text{Sailors}) \]
• Solution 2: \[ \pi_{sname}(\sigma_{bid=103}(\text{Reserves} \Join \text{Sailors})) \]

Find names of sailors who’ve reserved boat #103

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

Also called a logical query plan

Expressing an RA expression as a Tree
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:

Can also define Tempboats using union
Try the “AND” version yourself

What about aggregates?

- Extended relational algebra
- \( \varphi \) \( \text{age}, \text{avg}(\text{rating}) \rightarrow \text{avgr} \)

Relational Calculus (RC)

- RA is procedural
  - \( \pi \text{A}(\sigma \text{A}=a \text{R}) \) and \( \sigma \text{A}=a(\pi \text{A} \text{R}) \) are equivalent but different expressions
- RC
  - non-procedural and declarative
  - describes a set of answers without being explicit about how they should be computed

- TRC (tuple relational calculus)
  - variables take tuples as values
  - we will primarily do TRC
- DRC (domain relational calculus)
  - variables range over field values

TRC: example

\[
\{P | \exists S \in \text{Sailors} (S.\text{rating} > 7 \land P.\text{name} = S.\text{name} \land P.\text{age} = S.\text{age})\}
\]

- \( P \) is a tuple variable
  - with exactly two fields \text{name} and \text{age} (schema of the output relation)
  - \text{name} = S.\text{name} \land \text{age} = S.\text{age} gives values to the fields of an answer tuple
- Use parentheses, \( \exists \), \( \forall \), \( \land \), \( \lor \), \( < \), \( = \), etc as necessary
- \( A \Rightarrow B \) is very useful too

\( A \Rightarrow B \)

- A “implies” B
- Equivalently, if A is true, B must be true
- Equivalently, \( \neg A \lor B \), i.e.
  - either A is false (then B can be anything)
  - otherwise (i.e. A is true) B must be true
Useful Logical Equivalences

• \( \forall x \; P(x) = \neg \exists x \; \neg P(x) \)
  - There exists
  - For all
  - Logical AND
  - Logical OR
  - NOT

• \( \neg(P \lor Q) = \neg P \land \neg Q \)
• \( \neg(P \land Q) = \neg P \lor \neg Q \)
  - Similar, \( \neg(P \lor Q) = \neg P \land \neg Q \) etc.

• \( A \Rightarrow B = \neg A \lor B \)

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

\[ \{ P \mid \exists S \in \text{Sailors} \exists R1 \in \text{Reserves} \exists R2 \in \text{Reserves} (S.sid = R1.sid \land S.sid = R2.sid \land R1.bid \neq R2.bid) \land P.sname = S.sname \} \]

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 in RA

TRC: example

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

• Find the names of sailors who have reserved all red boats

\[ \{ P \mid \exists S \in \text{Sailors} \forall B \in \text{Boats} (\exists R \in \text{Reserves} (S.sid = R.sid \land R.bid = B.bid)) \land (P.sname = S.sname) \} \]

How will you change the previous TRC expression?
TRC: example

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

- Find the names of sailors who have reserved all red boats

Recall that A ⇒ B is logically equivalent to ¬ A ∨ B
so ⇒ can be avoided, but it is cleaner and more intuitive

DRC: example

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

- Find the name and age of all sailors with a rating above 7

TRC:
\( \{ P \mid \exists S \in \text{Sailors} (S.\text{rating} > 7 \land \text{P.name} = S.\text{name} \land \text{P.age} = S.\text{age}) \} \)

DRC:
\( \{ <N, A> \mid \exists <I, N, T, A> \in \text{Sailors} \land T > 7 \} \)

- Variables are now domain variables
- We will use TRC – both are equivalent
- Another option to write coming soon!

The famous “Beers” database

Bars
- Each has an address
- Serves Beers

Drinkers
- Each has an address
- Likes Beers

Beers
- Each has a brewer

“Beers” as a Relational Database

Bar
- Name
- Address
- Serves

Beer
- Name
- Brewer
- Price

Drinker
- Name
- Address
- Likes

Frequents
- Drinker
- Bar
- Times_a_week

Serves
- Bar
- Beer
- Likes

LIKE DRINKER CATEGORY 1

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

Drinker Category 1

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

Find drinkers that frequent some bars that serve some beer they like.

Drinker Category 2

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

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

Free HW question hint!

Drinker Category 3

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

Find drinkers that frequent only bars that serve 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.

Find drinkers that frequent only bars that serve 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.

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

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

Find drinkers that frequent only bars that serve 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 \( \vee \) and \( \Rightarrow \)
  - then you can systematically go to a "correct" SQL or RA query

Drinker Category 5!

From RC to SQL

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

\[
\{ x \mid \exists L \text{ Likes}(L.drinker = x.drinker) \land \exists S \text{ Serves}(L.beer = S.beer) \Rightarrow \exists F \text{ Frequents}([F.drinker = L.drinker] \land [F.bar = S.bar]) \} 
\]

Step 1: Replace \( \lor \) with \( \exists \) using de Morgan's Laws

\[
\exists x \forall y. \text{Likes}(x, y) \land \text{Serves}(y, \text{Beer}(x) = \text{Beer}(y)) \Rightarrow \exists F \text{ Frequents}([F.drinker = L.drinker] \land [F.bar = S.bar]) \}
\]

SQL or RA does not have \( \forall \)

Now you got all \( \exists \) and \( \forall \) expressible in RA/SQL

From RC to SQL (or RA)

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

\[
\{ x \mid \exists L \text{ Likes}(L.drinker = x.drinker) \land \exists S \text{ Serves}(L.beer = S.beer) \Rightarrow \exists F \text{ Frequents}([F.drinker = L.drinker] \land [F.bar = S.bar]) \} 
\]

Step 1: Replace \( \lor \) with \( \exists \) using de Morgan's Laws

\[
\exists x \forall y. \text{Likes}(x, y) \land \text{Serves}(y, \text{Beer}(x) = \text{Beer}(y)) \Rightarrow \exists F \text{ Frequents}([F.drinker = L.drinker] \land [F.bar = S.bar]) \}
\]

SQL or RA does not have \( \forall \)

Now you got all \( \exists \) and \( \forall \) expressible in RA/SQL

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 expressible in another
      - \( \exists S \) \( \leftarrow \left( S \in \text{Sailors} \right) \) -- infinitely many tuples -- an "unsafe" query
    - More when we do "Datalog", also see Ch. 4.4 in [RG]