Relational Algebra
and
Relational Calculus

Chapter 4
Formal Query Languages

- What is the basis of Query Languages?
- Two formal Query Languages form the basis of “real” query languages (e.g. SQL):
  - Relational Algebra: Operational, it provides a recipe for evaluating the query. Useful for representing execution plans.
  - Relational Calculus: Lets users describe what they want, rather than how to compute it. (Non-operational, declarative.)
What is an “Algebra”

- Set of operands and operations that they are “closed” under all compositions

- Examples
  - Boolean algebra - operands are the logical values True and False, and operations include AND(), OR(), NOT(), etc.
  - Integer algebra - operands are the set of integers, operands include ADD(), SUB(), MUL(), NEG(), etc. many of which have special in-fix operator symbols (+,-,*,-) 

- In our case operands are relations, what are the operators?
Example Instances

- “Sailors” and “Reserves” relations for our examples.

- We’ll use “named field notation”, which assumes 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>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S2</th>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td></td>
<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* ($\pi$) 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* ($\cup$) Tuples in reln. 1 and in reln. 2.

- Additional operations:
  - Intersection, *join*, 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?)

<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>

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

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

- Selects rows that satisfy \textit{selection condition}.
- No duplicates in result! (Why?)
- \textit{Schema} of result identical to schema of (only) input relation.
- \textit{Result} relation can be the \textit{input} for another relational algebra operation! (\textit{Operator composition}.)

\begin{table}
\begin{tabular}{|c|c|c|c|}
\hline
sid & sname & rating & age \\
\hline
28 & yuppy & 9 & 35.0 \\
31 & lubber & 8 & 55.5 \\
44 & guppy & 5 & 35.0 \\
58 & rusty & 10 & 35.0 \\
\hline
\end{tabular}
\end{table}

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

\begin{tabular}{|c|c|}
\hline
sname & rating \\
\hline
yuppy & 9 \\
rusty & 10 \\
\hline
\end{tabular}

\[ \pi_{\text{iname, rating}}(\sigma_{\text{rating}>8}(S2)) \]
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.
- What is the *schema* of result?

<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>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppie</td>
<td>9</td>
<td>35.0</td>
</tr>
</tbody>
</table>

$S_1 \cup S_2$

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<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>

$S_1 \setminus S_2$

<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>
</tbody>
</table>

$S_1 \cap S_2$
**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**: \( \rho(T(1 \rightarrow \text{sid}_1, 5 \rightarrow \text{sid}_2), S_1 \times R_1) \)
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>

\( S_l \bowtie S_l.sid < R_l.sid \ R_l \)

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

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

  Result schema similar to cross-product, but only one copy of fields for which equality is specified.

- **Natural Join**: Equijoin on *all* common fields (no labels on bowtie).
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 $y$:
  
  $A/B = \{ x \mid \exists (x, y) \in A \quad \forall (y) \in B \}$

  - i.e., $A/B$ contains all $x$ tuples (sailors) such that for every $y$ tuple (boat) in $B$, there is an $xy$ tuple in $A$.

  - 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>A</th>
<th>A/B1</th>
<th>A/B2</th>
<th>A/B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sno</td>
<td>pno</td>
<td>pno</td>
<td>pno</td>
</tr>
<tr>
<td>s1</td>
<td>p1</td>
<td>p2</td>
<td>p2</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
<td>p2</td>
<td>p4</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **B1**: p2
- **B2**: p2, p4
- **B3**: p1, p2, p4
Expressing A/B Using Basic Operators

- Division is not essential; it’s 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.

\[
\text{Disqualified x values: } \pi_x ((\pi_x (A) \times B) - A)
\]

\[
A/B: \quad \pi_x (A) - \text{disqualified tuples}
\]
Relational Algebra Examples

- Assume the following extended schema:
  - Sailors(sid: integer, sname: string, rating: integer, age: real)
  - Reserves(sid: integer, bid: integer, day: date)
  - Boat(bid: integer, bname: string, bcolor: string)

- Objective: Write a relational algebra expression whose result instance satisfies the specified conditions
  - May not be unique
  - Some alternatives might be more efficient (in terms of time and/or space)
Names of sailors who’ve reserved boat #103

Solution 1: \[ \pi_{\text{name}}((\sigma_{\text{bid}=103} \text{Reserves}) \bowtie \text{Sailors}) \]

Solution 2: \[ \rho (\text{Temp1}, \sigma_{\text{bid}=103} \text{Reserves}) \]
\[ \rho (\text{Temp2}, \text{Temp1} \bowtie \text{Sailors}) \]
\[ \pi_{\text{name}} (\text{Temp2}) \]

Solution 3: \[ \pi_{\text{name}}(\sigma_{\text{bid}=103} (\text{Reserves} \bowtie \text{Sailors})) \]
Names of sailors who’ve reserved a red boat

- Information about boat color only available in Boats; so need an extra join:

  \[ \pi_{sname}(\sigma_{\text{color} = 'red'}(\text{Boats} \bowtie \text{Reserves} \bowtie \text{Sailors})) \]

- A more efficient solution:

  \[ \pi_{sname}(\pi_{\text{sid}}(\pi_{\text{bid}}(\sigma_{\text{color} = 'red'}(\text{Boats} \bowtie \text{Res} \bowtie \text{Sailors})))) \]

A query optimizer can find this, given the first solution!
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:
  
  $$\rho (\sigma_{\text{color} = \text{'red'} \lor \text{color} = \text{'green'}}(\text{Boats}))$$

  $$\pi_{\text{sname}}(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors})$$

- Can also define Tempboats using union! (How?)
- What happens if $\lor$ is replaced by $\land$ in this query?
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):

\[
\rho (\text{Tempred}, \pi_{\text{sid}} ((\sigma \text{color='red'} \text{Boats}) \bowtie \text{Reserves}))
\]

\[
\rho (\text{Tempgreen}, \pi_{\text{sid}} ((\sigma \text{color='green'} \text{Boats}) \bowtie \text{Reserves}))
\]

\[
\pi_{\text{sname}}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors})
\]
Names of sailors who’ve reserved all boats

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

  \[
  \rho (\text{Temp} \varpi_{sid, bid} (\text{Reserves}) \div (\pi_{bid} \text{Boats}))
  \]

  \[
  \pi_{sname} (\text{Temp} \bowtie\text{Sailors})
  \]

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

  \[
  \ldots / \pi_{bid} (\sigma_{bname='Interlake'} \text{Boats})
  \]
Relational Calculus

- Comes in two flavors: **Tuple relational calculus** (TRC) and **Domain relational calculus** (DRC).
- Calculus has variables, constants, comparison ops, logical connectives and quantifiers.
  - **TRC**: Variables range over (i.e., get bound to) tuples.
  - **DRC**: Variables range over domain elements (= field values).
  - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called *formulas with unbound formal variables*. An answer tuple is essentially an assignment of constants to these variables that make the formula evaluate to *true*. 
A Fork in the Road

- TRC and DRC are semantically similar
- In TRC, tuples share an equal status as variables, and field referencing can be used to select tuple parts
- In DRC, formal variables are explicit
- In the book you will find extensive discussions and examples of TRC Queries (Sections 4.3.1) and a lesser treatment of DRC.
- To even things out, in this lecture I will focus on DRC examples
Domain Relational Calculus

- **Query** has the form:
  \[
  \{<x_1, x_2, \ldots, x_n> | p(<x_1, x_2, \ldots, x_n>)\}
  \]

- **Answer** includes all tuples \(<x_1, x_2, \ldots, x_n>\) that make the formula \(p(<x_1, x_2, \ldots, x_n>)\) true.

- **Formula** is recursively defined, starting with simple *atomic formulas* (getting tuples from relations or making comparisons of values), and building bigger and better formulas using the *logical connectives*.
DRC Formulas

- **Atomic formula:**
  - \(<x_1,x_2,\ldots,x_n> \in R_{name}, \text{ or } X \ op \ Y, \text{ or } X \ op \ \text{constant}\)
  - \(\text{op}\) is one of \(<,>,=,\leq,\geq,\neq\)

- **Formula:**
  - an atomic formula, \ or
  - \(\neg p, p \land q, p \lor q\), where \(p\) and \(q\) are formulas, \ or
  - \(\exists X (p(X))\), where variable \(X\) is free in \(p(X)\), \ or
  - \(\forall X (p(X))\), where variable \(X\) is free in \(p(X)\)

\(\exists X(p(X))\) is read as “there exists a setting of the variable \(X\) such that \(p(X)\) is true”. \(\forall X(p(X))\) is read as “for all values of \(X\), \(p(X)\) is true”
Free and Bound Variables

- The use of quantifiers $\exists X$ and $\forall X$ in a formula is said to bind $X$.
  - A variable that is not bound is free.
- Let us revisit the definition of a query:
  \[
  \{<x_1, x_2, \ldots, x_n> \mid p(<x_1, x_2, \ldots, x_n>)\}
  \]
- There is an important restriction: the variables $x_1, \ldots, x_n$ that appear to the left of ‘|’ must be the only free variables in the formula $p(\ldots)$. 
Examples

- Recall the example relations from last lecture

| Sailors: | | Reservations: | | Boats: |
|----------|-----------------|-----------------|-----------------|
| sid | sname | rating | age | sid | bid | day | bid | bname | color |
| 22 | Dustin | 7 | 45.0 | 22 | 101 | 10/10/98 | 101 | Interlake | blue |
| 29 | Brutus | 1 | 33.0 | 22 | 102 | 10/10/98 | 102 | Interlake | red |
| 31 | Lubber | 8 | 55.5 | 22 | 103 | 10/8/98 | 103 | Clipper | green |
| 32 | Andy | 8 | 25.5 | 22 | 104 | 10/7/98 | 104 | Marine | red |
| 58 | Rusty | 10 | 35.0 | 31 | 102 | 11/10/98 | | |
| 64 | Horatio | 7 | 35.0 | 31 | 103 | 11/6/98 | | |
| 71 | Zorba | 10 | 16.0 | 31 | 104 | 11/12/98 | | |
| 74 | Horatio | 9 | 35.0 | 64 | 101 | 9/5/98 | | |
| 85 | Art | 3 | 25.5 | 64 | 102 | 9/8/98 | | |
| 95 | Bob | 3 | 63.5 | 74 | 103 | 9/8/98 | | |
Find sailors with ratings > 7

\{<I,N,T,A> | <I,N,T,A> \in \text{Sailors} \land T > 7\}

- The condition \(<I,N,T,A> \in \text{Sailors}\) binds the domain variables \(I, N, T\) and \(A\) to fields of any Sailors tuple.
- The term, \(<I,N,T,A>\), to the left of ‘|’ (which should be read as \textit{such that}) says that every tuple, that satisfies \(T > 7\) is in the answer.
- Modify this query to answer:
  - Find sailors who are older than 18 or have a rating under 9, and are called ‘Joe’.
Same query using TRC

- Find all sailors with ratings above 7

  \{S \mid S \in \text{Sailors} \land S.\text{rating} > 7\}

- Note, here \(S\) is a tuple variable

  \{X \mid S \in \text{Sailors} \land S.\text{rating} > 7 \land X.\text{name} = S.\text{name} \land X.\text{age} = S.\text{age}\}

- Here \(X\) is a tuple variable with 2 fields (name, age). This query implicitly specifies projection (\(\pi\)) and renaming (\(\rho\)) relational algebra operators
Sailors rated > 7 who reserved boat #103

\{<I,N,T,A> | <I,N,T,A> \in Sailors \land T > 7 \land \\
\exists \ I_r, \ B_r, \ D(<I_r, \ B_r, \ D> \in Reserves \land \\
\ I_r = I \land \ B_r = 103)\}\}

- We have used \(\exists \ I_r, \ B_r, \ D(\ldots)\) as a shorthand for
  \(\exists \ I_r(\exists \ B_r(\exists \ D(\ldots)))\)
- Note the use of \(\exists\) to find a tuple in Reserves
  that ‘joins with’ (\(\bowtie\)) the Sailors tuples under
  consideration.
Find sailors rated > 7 who’ve reserved a red boat

\{<I,N,T,A> | <I,N,T,A> \in \text{Sailors} \land T > 7 \land \\
\exists I_r, Br, D(<I_r, Br, D> \in \text{Reserves} \land I_r = I \land \\
\exists B, Bn, C(<B, Bn, C> \in \text{Boats} \land B = Br \land C = \text{‘red’})\}\}

- Observe how the parentheses control the scope of each quantifier’s binding.
- This may look cumbersome, but with a good user interface, it is very intuitive. (MS Access, QBE)
Names of all Sailors who have reserved boat 103

\[ \{ \langle N \rangle | \exists I, T, A (\langle I, N, T, A \rangle \in Sailor) \]
\[ \land \exists I_r, Br, D (\langle I_r, Br, D \rangle \in Reserves \land I_r = I \land Br = 103) \} \]

- Note that only the sname field is retained in the answer and that only \( N \) is a free variable.
- A more compact version

\[ \{ \langle N \rangle | \exists I, T, A (\langle I, N, T, A \rangle \in Sailor) \]
\[ \land \exists D (\langle I, 103, D \rangle \in Reserves) \} \]
Recall how queries of this type used of the “division” operator in relational algebra.

The trick is that we use “for all” quantification (\(\forall\)) in place of “there exists” quantification (\(\exists\)).

Domains of variables are determined when they are bound.

Think of it as considering each variable’s “domain” of independently in our substitution.

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>101</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>101</td>
<td>Interlake</td>
<td>green</td>
</tr>
<tr>
<td>101</td>
<td>Clipper</td>
<td>blue</td>
</tr>
<tr>
<td>101</td>
<td>Clipper</td>
<td>red</td>
</tr>
<tr>
<td>101</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>101</td>
<td>Marine</td>
<td>blue</td>
</tr>
<tr>
<td>101</td>
<td>Marine</td>
<td>red</td>
</tr>
<tr>
<td>101</td>
<td>Marine</td>
<td>green</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>marine</td>
<td>red</td>
</tr>
</tbody>
</table>
Sailors who’ve reserved all boats

\[
\left\{ (I, N, T, A) \mid (I, N, T, A) \in \text{Sailors} \land \\
\forall \ B, BN, C \left( \neg (B, BN, C) \in \text{Boats} \right) \lor \\
\exists Ir, Br, D \left( (Ir, Br, D) \in \text{Reserves} \land I = Ir \land Br = B \right) \right\}
\]

- Find all sailors \( I \) such that for each 3-tuple \( (B, BN, C) \) either it is not a tuple in Boats or there is a tuple in Reserves showing that sailor \( I \) has reserved it.
Find sailors who’ve reserved all boats (again!)

\[
\{ \langle I, N, T, A \rangle \mid \langle I, N, T, A \rangle \in \text{Sailors} \wedge \\
\forall \langle B, BN, C \rangle \in \text{Boats} \\
\exists \langle Ir, Br, D \rangle \in \text{Reserves} (I = Ir \wedge Br = B) \}
\]

- Simpler notation, same query. (Much clearer!)
- To find sailors who’ve reserved all red boats:
  
  ..... \[
  \{ C \neq 'red' \vee \exists \langle Ir, Br, D \rangle \in \text{Reserves} (I = Ir \wedge Br = B) \}
  \]
Unsafe Queries, Expressive Power

- It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called unsafe.
  - e.g., \[
  \{ <I,N,T,A> \mid <I,N,T,A> \notin \text{Sailors} \}\]

- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.

- **Relational Completeness:** Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.
Summary

- Relational calculus is non-operational, and users define queries in terms of what they want, not in terms of how to compute it. (Declarativeness.)
- Algebra and safe calculus have same expressive power, leading to the notion of \textit{relational completeness}.