Relational Algebra and Calculus

Chapter 4

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 implementation:

1. **Relational Algebra**: More procedural, very useful for representing execution plans.
2. **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-procedural, declarative.)

☛ Understanding Algebra & Calculus is key to understanding SQL and query processing!

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.
Example Instances

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

<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>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

- **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 relation 1 or in relation 2.
Relational Algebra (contd.)

- Additional operations:
  - Intersection, join, division, renaming.
  - Not essential, because can be implemented using the five basic operations.
  - 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?)
Projection

<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>

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

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

Selection

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

Database Management Systems, R. Ramakrishnan and J. Gehrke
**Selection**

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

<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>

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

**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>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>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ S1 \cup S2 \]

\[ S1 \cap S2 \]
Cross-Product

- Each row of the one relation is paired with each row of the other relation.
- Result schema has one field per field of both input relations, with field names ‘inherited’ if possible.
- In the result, there may be two fields with the same name, e.g. both S1 and R1 have a field called sid.
- Then, apply the renaming operator, e.g.
  \[ \rho \left( C(1 \rightarrow \text{sid}1, 5 \rightarrow \text{sid}2) \right), \ S1 \times R1 \]

Cross-Product

<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>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</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>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</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>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</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>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>
Joins

- **Condition Join**: \( R \bowtie_c S = \sigma_c (R \times S) \)

<table>
<thead>
<tr>
<th>(sid)</th>
<th>surname</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>

\[ S_l \bowtie S_l \text{.sid < } R_l \text{.sid} \]

- 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*.

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</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>

\[ S_l \bowtie_{\text{sid}} R_l \]

- Result schema similar to cross-product, but only one copy of fields for which equality is specified.
- **Natural Join**: Equijoin on *all* common fields.
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 $y$:
  - $A/B = \{ \{x\} \mid \exists \{x, y\} \in A \land \{y\} \in B \}$
  - i.e., $A/B$ contains all $x$ tuples (sailors) such that for every $y$ tuple (boat) in $B$, there is an $xy$ tuple (reservation) in $A$.

- 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

| sno | pno | | sno | pno | | sno | pno |
|-----|-----| |     |     | |     |     | |     |     |
| s1  | p1  | | p2  | p2  | | p1  | p1  | | s1  | p1  |
| s1  | p2  | | p2  | p4  | | p2  | p2  | | s2  | p2  |
| s1  | p3  | | p4  |     | | p4  | p4  | | s4  | p4  |
| s2  | p1  | |     |     | |     |     | | s1  | p1  |
| s2  | p2  | |     |     | |     |     | | s2  | p2  |
| s3  | p2  | |     |     | |     |     | | s3  | p2  |
| s4  | p2  | |     |     | |     |     | | s4  | p2  |
| s4  | p4  | |     |     | |     |     | | s4  | p4  |

\[ A \quad A/B1 \quad A/B2 \quad A/B3 \]
Expressing $A/B$ Using Basic Operators

- Division is not essential op; 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 $A$.

  \[ \pi_x (\pi_x(A) \times B) - A \]

  Disqualified $x$ values:

  \[ \pi_x (\pi_x(A) \times B) - A \]

  $A/B$: \[ \pi_x (A) \] — all disqualified tuples

Find names of sailors who’ve reserved boat #103

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

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

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

- Information about boat color only available in Boats; so need an extra join:
  \[ \pi_{sname}(\sigma_{\text{color} = \text{red}}(\text{Boats}) \bowtie \text{Reserves} \bowtie \text{Sailors}) \]

- A more efficient solution:
  \[ \pi_{sname}(\pi_{\text{sid}}(\sigma_{\text{color} = \text{red}}(\text{Boats}) \bowtie \text{Res} \bowtie \text{Sailors})) \]

- A query optimizer can find this given the first solution!

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:
  \[ \rho(\pi_{\text{color} = \text{red} \lor \text{color} = \text{green}}(\text{Tempboats} \bowtie \text{Boats})) \]

- Can also define Tempboats using union! (How?)

- What happens if \( \lor \) is replaced by \( \land \) in this query?
Find 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):

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

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

\[
\pi_{\text{sname}}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors})
\]

Find the names of sailors who’ve reserved all boats

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

\[
\rho (\text{Tempsids, } (\pi_{\text{sid,bid}} \text{Reserves}) / (\pi_{\text{bid}} \text{Boats}))
\]

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

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

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

- Comes in two flavours: *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*. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.

Domain Relational Calculus

- Query has the form: \( \{x_1, x_2, \ldots, x_n \mid p(x_1, x_2, \ldots, x_n) \} \)
- Answer includes all tuples \( \{x_1, x_2, \ldots, x_n \} \) that make the formula \( p(x_1, x_2, \ldots, x_n) \) be 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:**
  - \(\langle x_1, x_2, \ldots, x_n \rangle \in R_{\text{name}}\), or
  - \(X \; \text{op} \; Y\), or \(X \; \text{op} \; \text{constant}\)
  - \(\text{op}\) 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 \(p(X)\)

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:
  \[
  \{\langle x_1, x_2, \ldots, x_n \rangle \mid p(\langle x_1, x_2, \ldots, x_n \rangle)\}
  \]

- 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 \(p(...).\)
Find all sailors with a rating above 7

\[
\{ \langle I,N,T,A \rangle | \langle I,N,T,A \rangle \in \text{Sailors} \land T > 7 \}
\]

- The condition \( \langle I,N,T,A \rangle \in \text{Sailors} \) ensures that the domain variables \( I, N, T \) and \( A \) are bound to fields of the same Sailors tuple.
- The term \( \langle I,N,T,A \rangle \) to the left of `\(|` (which should be read as such that) says that every tuple \( \langle I,N,T,A \rangle \) that satisfies \( T > 7 \) is in the answer.
- How to find sailors who are older than 18 or have a rating under 9, and are called ‘Joe’?

Find sailors rated > 7 who’ve reserved boat #103

\[
\{ \langle I,N,T,A \rangle | \langle I,N,T,A \rangle \in \text{Sailors} \land T > 7 \land \\
\exists Ir, Br, D \ (\langle Ir, Br, D \rangle \in \text{Reserves} \land Ir = I \land Br = 103) \}
\]

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

\[
(I,N,T,A), (I,N,T,A) \in \text{Sailors} \land T > 7 \land \\
\exists I,r,Br,D \ (I,r,Br,D) \in \text{Reserves} \land I = r \land \\
\exists B, BN, C \ ((B, BN, C) \in \text{Boats} \land B = Br \land C = \text{‘red’})
\]

❖ 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. (Wait for QBE!)

Find sailors who’ve reserved all boats

\[
(I,N,T,A) 
\forall B, BN, C \ (B, BN, C) \in \text{Boats} \\
\exists I,r,Br,D \ (I,r,Br,D) \in \text{Reserves} \land I = r \land B = B
\]

❖ 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!)

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

- Simpler notation, same query. (Much clearer!)
- To find sailors who’ve reserved all red boats:

\[
\ldots \quad \{ C \neq 'red' \lor \exists \langle Ir, Br, D \rangle \in \text{Reserves} \{ I = Ir \land Br = B \} \}
\]

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., \( |S| \neg (S \in \text{Sailors}) \)
- 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/safe relational calculus.
Summary

❖ The relational model has rigorously defined query languages that are simple and powerful.
❖ Relational algebra is more procedural; useful as internal representation for query evaluation plans.
❖ Several ways of expressing a given query; a query optimizer should choose the most efficient version.

Summary (contd.)

❖ Relational calculus is non-procedural, 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 relational completeness.
❖ All practical query languages should be relationally complete.