Relational Query Languages

- A major strength of the relational model: supports simple, powerful querying of data.
- Relational query languages:
  - High-level declarative: say “what you want” not “how you get it”
  - Based on a formal mathematical model.
  - 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:
  - Relational Algebra: Operational, very useful for representing execution plans.
  - Relational Calculus: Declarative (lets users describe what they want, rather than how to compute it), useful for representing query semantics.
Preliminaries

- A query is applied to relation instances. The result of a query is also a relation instance.
  - Schemas of input relations for a query are fixed.
  - Schema for the result of a given query is also fixed.
    - Determined by definition of query language constructs.
- How to name fields in queries?
  - Positional notation easier for formal definitions.
  - Named-field notation more readable.
  - Both used in SQL.

Relational Algebra

- Basic operations:
  - Selection (σ) Selects a subset of rows from relation.
  - Projection (π) Deletes unwanted columns from relation.
  - Cross-product (×) Allows us to combine two relations.
  - Set-difference (−) Tuples in reln. 1, but not in reln. 2.
  - Union (∪) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, join, division, renaming
  - Not essential, but (very!) useful.

Closure Property

- Each operation takes one or more relations and returns a relation.
  - Data model for input and output of an operation is relation.
  - Algebra is closed with respect to the data model.
- Given closure property, operations can be composed!
Example Instances

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

<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>

Projection

- Deletes attributes that are not in projection list.
- Schema of result contains exactly the fields in the projection list.
- 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>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>yuppy</td>
<td>9</td>
</tr>
<tr>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>guppy</td>
<td>5</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.0</td>
</tr>
<tr>
<td>55.5</td>
</tr>
</tbody>
</table>

Selection

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

<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>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{rating}}(\sigma_{\text{rating}>8}(S2)) \]
Union, Intersection, Set-Difference

- Set operations:
  - Union
  - Intersection
  - Set-Difference
- All of them 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>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>

Example Set Operations

<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>
<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>

**SIUS2**

<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>

**SI\(\cap\)S2**

<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>
</tbody>
</table>

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>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>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>31</td>
<td>103</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
<td>31</td>
<td>103</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
<td>58</td>
<td>103</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>

* Renaming operator: \(\rho\) (\(C(1 \rightarrow \text{sid}1, 5 \rightarrow \text{sid}2), \text{S1} \times \text{R1})\)
Joins

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

<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>

- **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>sname</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>31</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- **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 \( y \); let \( B \) have only field \( y \):
  - \( A/B = \left\{ x \right\} \in A \exists \left\{ y \right\} \in B \)
  - i.e., \( A/B \) contains all \( x \) tuples (sailors) such that for every \( y \) tuple (boat) in \( B \), there is an \( xy \) tuple in \( A \).
  - Or: 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>s1</th>
<th>p1</th>
<th>pno</th>
<th>B1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p2</td>
<td>p2</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
<td>p4</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
<td></td>
<td>B2</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>

$A$  $A/B1$  $A/B2$  $A/B3$

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 $B$.
  - Disqualified $x$ values: $\pi_x (\pi_x (A) \times B) - A$
  - $x$ value is disqualified if by attaching $y$ value from $B$, we obtain an $xy$ tuple that is not in $A$.

Find names of sailors who’ve reserved boat #103

- Solution 1: $\pi_{sname}(\sigma_{bid=103}(Reserves) \bowtie Sailors)$
- Solution 2: $\rho (Temp1, \sigma_{bid=103}(Reserves))$
  - $\rho (Temp1, \sigma_{bid=103}(Reserves))$
  - $\pi_{sname}(Temp2)$
- Solution 3: $\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$
  - Algebraic equivalences!
Find names of sailors who’ve reserved a red boat

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

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:
  \[ \rho (\text{Tempboats}, (\sigma_{\text{color} = \text{red}} \lor \text{color} = \text{green} \text{Boats})) \]
  \[ \pi_{\text{name}}(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors}) \]

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

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):
  \[ \rho (\text{Tempred}, \pi_{\text{sid}}((\sigma_{\text{color} = \text{red}} \text{Boats}) \bowtie \text{Reserves})) \]
  \[ \rho (\text{Tempgreen}, \pi_{\text{sid}}((\sigma_{\text{color} = \text{green}} \text{Boats}) \bowtie \text{Reserves})) \]
  \[ \pi_{\text{name}}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors}) \]
Find the names of sailors who’ve reserved all boats

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

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

  \[ \pi_{\text{name}}(\text{Tempsid} \bowtie \text{Sailors}) \]

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

  \[ \sigma_{\text{name}} (\bowtie_{\text{bid}} \text{name} = \text{'Interlake'} \text{Boats}) \]

Relational Calculus

- Relational Calculus uses variables, constants, comparison ops, logical connectives and quantifiers.
- Two forms: Tuple relational calculus (TRC) and Domain relational calculus (DRC).
  - TRC: Variables range over (i.e., get bound to) tuples.
  - DRC: Variables range over domain elements (field values).
- Both are simple subsets of first-order logic.
- Formulas: expressions in the calculus.
- An answer tuple: an assignment of constants to variables that make the formula evaluate to true.

Domain Relational Calculus

- Query has the form:

  \[ p(x_1, x_2, \ldots, x_n) \]

- 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:**
  - \[ \{x_1, x_2, \ldots, x_n\} \in \text{Rname} \text{, or } X \text{ op } Y \text{, or } X \text{ op constant} \]
  - \text{op} \text{ is one of } \{<, >, =, \leq, \geq, \neq\}

- **Formula:**
  - an atomic formula, or
  - \( p \lor q \), 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 \( p(X) \)

- The use of quantifiers \( \exists X \) and \( \forall X \) is said to **bind** \( X \).
  - A variable that is not bound is free.

### 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:
  \[
  \{x_1, x_2, \ldots, x_n\} \models p\{x_1, x_2, \ldots, x_n\}
  \]

- There is an important restriction: the variables \( x_1, ..., x_n \) that appear to the left of `|` must be the only free variables in the formula \( p(...) \).

### Find all sailors with a rating above 7

- The condition \( \{I, N, T, A\} \in \text{Sailors} \land T > 7 \) ensures that the domain variables \( I, N, T \) and \( A \) are bound to fields of the same Sailors tuple.

- The term \( \{I, N, T, A\} \) to the left of `|` (which should be read as such that) says that every tuple \( \{I, N, T, A\} \) 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'.
Find sailors who have reserved boat #103

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

- We have used \( \exists I, Br, D \ (\ldots) \) as a shorthand for \( \exists I (\exists Br (\exists D (\ldots))) \)
- Note the use of \( \exists \) to find a tuple in Reserves that 'joins with' the Sailors tuple under consideration.

Find sailors who have reserved a red boat

\[ \{ I, N, T, A \} \land (I, N, T, A) \in \text{Sailors} \land T > 7 \land \exists I, Br, D \ (I, Br, D) \in \text{Reserves} \land I \in \text{I} \land \exists B, BN, C \ (B, BN, C) \in \text{Boats} \land B = Br \land C = \text{red} \]

- 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)

Find sailors who’ve reserved all boats

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

- 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!)

\[\forall (B, BN, C) \in \text{Boats} \exists (I, Br, D) \in \text{Reserves} \mid I = Ir \land Br = B\]

- Simpler notation, same query. (Much clearer!)
- To find sailors who’ve reserved all red boats:
  \[\forall (C \neq \text{red}) \exists (I, Br, D) \in \text{Reserves} \mid I = Ir \land Br = B\]

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., \[I \in \text{Sailors} \land I \in \text{Reserves}\]
- 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

- The relational model has rigorously defined query languages that are simple and powerful.
- Relational algebra is more operational; 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-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 relational completeness.

Questions