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:

- **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.)
  - Understanding Algebra & Calculus is key to understanding SQL, 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 Relational Algebra and 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.

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

### S1
<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>

### S2
<table>
<thead>
<tr>
<th>sid</th>
<th>name</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

- 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, **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?)

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.)

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?

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.

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

Natural Join: Equijoin on all common fields.

Joins
- Condition join: \( R \bowtie^c S = \sigma_c (R \times S) \)

Equi-Join: A special case of condition join where the condition \( c \) contains only equalities and ^.

Natural Join: Equijoin on all common fields.
Find names of sailors who’ve reserved boat #103

- Solution 1: \( \pi_{\text{name}}(\sigma_{\text{bid}=103} \text{Reserves}) \bowtie \text{Sailors} \)
- Solution 2: \( \rho (\text{Temp}, \sigma_{\text{bid}=103} \text{Reserves}) \)
  \( \rho (\text{Temp2}, \text{Temp} \bowtie \text{Sailors}) \)
  \( \pi_{\text{name}}(\text{Temp2}) \)
- Solution 3: \( \pi_{\text{name}}(\sigma_{\text{bid}=103} (\text{Reserves} \bowtie \text{Sailors})) \)

Find names of sailors who’ve reserved a red boat

- Information about boat color only available in Boats; so need an extra join:
  \( \pi_{\text{name}}(\sigma_{\text{color}=\text{red}} \text{Boats}) \bowtie \text{Reserves} \bowtie \text{Sailors} \)
- A more efficient solution:
  \( \pi_{\text{name}}(\sigma_{\text{sid}}(\sigma_{\text{bid}}(\sigma_{\text{color}=\text{red}} \text{Boats}) \bowtie \text{Reserves} \bowtie \text{Sailors})) \)
  \( \bowtie \text{Reserves} \bowtie \text{Sailors} \)
  \( \pi_{\text{name}}(\text{Temp} \bowtie \text{Reserves} \bowtie \text{Sailors}) \)

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:
  \( \rho (\text{Tempboats}, (\sigma_{\text{color}=\text{red}} \cup \text{color}=\text{green} \text{Boats})) \)
  \( \pi_{\text{name}}(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors}) \)
- Can also define Tempboats using union! (How?)
- What happens if \( \cup \) is replaced by \( \cap \) 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 \( \text{sid} \) is a key for Sailors):
  \( \rho (\text{Tempred}, \pi_{\text{sid}}((\sigma_{\text{color}=\text{red}} \text{Boats}) \bowtie \text{Reserves})) \)
  \( \rho (\text{Tempgreen}, \pi_{\text{sid}}((\sigma_{\text{color}=\text{green}} \text{Boats}) \bowtie \text{Reserves})) \)
  \( \pi_{\text{name}}(\text{Tempred} \cap \text{Tempgreen} \bowtie \text{Sailors}) \)

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**. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to **true**.
Tuple Relational Calculus

- A query has the form: \{ T | p(T) \}
- Answer includes all tuples T that make the formula p(T) 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.

TRC Formulas

- Atomic formula:
  - R \in Rel, or R.a op S.b, or R.a op constant
  - op is one of \{<, >, =, \leq, \geq\}
- 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 \(\forall X\) and \(\exists X\) in a formula is said to bind X.
  - A variable that is not bound is free.
- Let us revisit the definition of a query: \{ T | p(T) \}
- There is an important restriction: the variable T that appears to the left of `|' must be the only free variable in the formula p(...).

Find all sailors with a rating above 7

- \{ S | S \in Sailors \land S.rating > 7 \}
- Query is evaluated on an instance of Sailors
- Tuple variable S is instantiated to each tuple of this instance in turn, and the condition “S.rating > 7” is applied to each such tuple.
- Answer contains all instances of S (which are tuples of Sailors) satisfying the condition.

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

- \{ S | (S \in Sailors) \land (S.rating > 7) \land (\exists R \in Reserves (R.sid = S.sid \land R.bid = 103)) \}
- Note the use of \(\exists\) to find a tuple in Reserves that ‘joins with’ the Sailors tuple under consideration.
- R is bound, S is not

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 | S \in 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

- 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.
- Relational calculus is non-operational, and users define queries in terms of what they want, not in terms of how to compute it. (Declarativeness.)
- Several ways of expressing a given query; a query optimizer should choose the most efficient version.
- Algebra and safe calculus have same expressive power, leading to the notion of relational completeness.

Nested Relations

- Attributes can be scalar (as before) or relation-valued
- Definition is recursive
- Example:
  
  ```
  create table Book (title: string, author:string, copies: (publ: string, pages: integer, date: integer))
  ```

  "copies" is a relation-valued field

Nested Relations

- A spectrum of algebras can be defined
- At one end:
  - Relational algebra plus `nest (v)` and `unnest (μ)`:
  
  ```
  title: string, author: string, copies: (publ: string, pages: integer, date: integer)
  ```

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
<th>copies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moby Dick</td>
<td>Melville</td>
<td>Prentice Hall</td>
</tr>
<tr>
<td></td>
<td></td>
<td>613 1971</td>
</tr>
<tr>
<td></td>
<td></td>
<td>McGraw Hill</td>
</tr>
<tr>
<td></td>
<td></td>
<td>542 1942</td>
</tr>
<tr>
<td>Maroon</td>
<td>Scott</td>
<td>{}</td>
</tr>
</tbody>
</table>

  Nulls must be supported in algebra

- At other end of spectrum:
  - `Selection` allows set comparators and constants and use of `select`, `project` inside the formula
  - `Projection` allows arbitrary NF2 algebra expression in addition to simple field names
  - `Union, difference`: recursive definitions
  - `Cross product`: usually just relational.

  Example: retrieve title, number of pages of all books by Melville:

  ```
  π(title, ⌈pages⌋[copies])[@author='Melville'](B)
  ```

Nested Relations Summary

- An early step on the way to OODBMS
- No products, only prototypes, but:
  - Many ideas from NF2 relations have survived
  - Collection types in SQL3 (nesting, unnesting)
  - Algebra ideas useful for Object Database QP
- Can provide a more natural model of data
  - Are a straightforward extension of relations:
    - many solutions are thus also straightforward
    - formal foundation of relational model remains