Relational Algebra and SQL

Johannes Gehrke
johannes@cs.cornell.edu
http://www.cs.cornell.edu/johannes

Slides from
Database Management Systems, 3rd Edition,
Ramakrishnan and Gehrke.
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.
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*. )
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>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</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>

<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>

<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>

<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>
Relational Algebra
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.

♦ 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>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>

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

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

\[
\pi \text{ age}(S2)
\]
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}.)

\[
\begin{array}{|c|c|c|c|}
\hline
\text{sid} & \text{sname} & \text{rating} & \text{age} \\
\hline
28 & \text{yuppy} & 9 & 35.0 \\
58 & \text{rusty} & 10 & 35.0 \\
\hline
\end{array}
\]

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

\[
\pi_{\text{sname, rating}}(\sigma_{\text{rating} > 8}(S2))
\]
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>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>44</td>
<td>gumpy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
</tbody>
</table>

**$S1 \cup S2$**

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

**$S1 - S2$**

<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>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:** \(\rho (C(1 \rightarrow \text{sid1}, 5 \rightarrow \text{sid2}), S1 \times R1)\)
Joins

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

<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>

\( S1 \Join S1.sid < R1.sid \)

- 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>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>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

  \[ S_1 \bowtie_{sid} R_1 \]

- **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 $y$:
  
  - $A/B = \{ \langle x \rangle \mid \forall \langle y \rangle \in B \exists \langle x, y \rangle \in A \}$

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

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
</tr>
<tr>
<td>s3</td>
</tr>
<tr>
<td>s4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>A/B1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>A/B2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>A/B3</th>
</tr>
</thead>
</table>
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 \( A \).

Disqualified \( x \) values:

\[ A/B: \]
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:_ \( \rho (\text{Temp1}, \sigma_{\text{bid}=103} \text{Reserves}) \)

\( \rho (\text{Temp2}, \text{Temp1} \bowtie \text{Sailors}) \)

\( \pi_{\text{name}} (\text{Temp2}) \)

_solution 3:_ \( \pi_{\text{name}} (\sigma_{\text{bid}=103} (\text{Reserves} \bowtie \text{Sailors})) \)
Find names of sailors who’ve reserved a red boat

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

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

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:

\[ ρ (Tempboats, (σ_{color = 'red' ∨ color = 'green'} Boats)) \]

\[ π_{sname}(Tempboats ⋈ Reserves ⋈ Sailors) \]

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

❖ What happens if ∨ is replaced by ∧ 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):

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

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

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

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

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

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

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

\[
..... / \pi_{\text{bid}} (\sigma \text{bname} = '\text{Interlaker'} \text{Boats})
\]
SQL
Basic SQL Query

SELECT [DISTINCT] target-list
FROM relation-list
[WHERE condition]

- Default is that duplicates are not eliminated!
  - Need to explicitly say “DISTINCT”

SELECT S.Name
FROM Sailors S
WHERE S.Age > 25

SELECT DISTINCT S.Name
FROM Sailors S
WHERE S.Age > 25
### SQL Query

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

<table>
<thead>
<tr>
<th>Sailors</th>
<th>Reserves</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>sid</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td></td>
</tr>
</tbody>
</table>

<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>
Conceptual Evaluation Strategy

• Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list
  - Discard resulting tuples if they fail condition.
  - Delete attributes that are not in target-list
  - If DISTINCT is specified, eliminate duplicate rows.

• This strategy is probably the least efficient way to compute a query!
  - An optimizer will find more efficient strategies to compute the same answers.
Example of Conceptual Evaluation

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

<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>
A Slightly Modified Query

SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103

• Would adding DISTINCT to this query make a difference?
Find sid’s of sailors who’ve reserved a red or a green boat

SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND (B.color='red' OR B.color='green')

SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='green'
What does this query compute?

SELECT  S.sid
FROM    Sailors S, Boats B1, Reserves R1, Boats B2, Reserves R2
WHERE   S.sid=R1.sid AND R1.bid=B1.bid AND
        S.sid=R2.sid AND R2.bid=B2.bid AND
        B1.color='red' AND B2.color='green'
Find sid’s of sailors who’ve reserved a red and a green boat

Key field!

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='green'
```

- **What if** `INTERSECT` **were replaced by** `EXCEPT`?
  - `EXCEPT` is set difference
Expressions and Strings

- Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.
- AS is used to name fields in result.
- LIKE is used for string matching
  - `_' stands for any one character
  - `%' stands for 0 or more arbitrary characters.

```
SELECT  S.age, S.age-5 AS age2, 2*S.age AS age2
FROM    Sailors S
WHERE   S.sname LIKE 'B_%B'
```
Nested Queries (with Correlation)

Find names of sailors who have reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
               FROM Reserves R
               WHERE R.bid=103 AND S.sid=R.sid)
```
**Nested Queries (with Correlation)**

Find names of sailors who have *not* reserved boat #103:

```
SELECT  S.sname
FROM    Sailors S
WHERE   NOT EXISTS (SELECT *
                     FROM   Reserves R
                     WHERE  R.bid=103 AND S.sid=R.sid)
```
Division in SQL

Find sailors who've reserved all boats

```
SELECT  S.sname
FROM    Sailors S
WHERE   NOT EXISTS ((SELECT  B.bid
                      FROM    Boats B)
                      EXCEPT
                      (SELECT  R.bid
                      FROM    Reserves R
                      WHERE   R.sid=S.sid))
```
Find sailors who’ve reserved all boats.

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
                   FROM Boats B
                   WHERE NOT EXISTS (SELECT R.bid
                                      FROM Reserves R
                                      WHERE R.bid=B.bid
                                      AND R.sid=S.sid))
```

Sailors $S$ such that ...

... there is no boat $B$ without ...

... a Reserves tuple showing $S$ reserved $B$
More on Set-Comparison Operators

- $op$ ANY, $op$ ALL
  - $op$ can be $>,<,=,\geq,\leq,\neq$

Find sailors whose rating is greater than that of all sailors called Horatio:

```
SELECT *  
FROM  Sailors S  
WHERE  S.rating > ALL (SELECT S2.rating  
FROM  Sailors S2  
WHERE  S2.sname='Horatio')
```
Aggregate Operators

- Significant extension of relational algebra.

- COUNT (*)
- COUNT ([DISTINCT] A)
- SUM ([DISTINCT] A)
- AVG ([DISTINCT] A)
- MAX (A)
- MIN (A)

- SELECT COUNT (*)
  FROM Sailors S

- SELECT AVG (S.age)
  FROM Sailors S
  WHERE S.rating=10

- SELECT COUNT (DISTINCT S.rating)
  FROM Sailors S
  WHERE S.sname=’Bob’
Find name and age of the oldest sailor(s) with rating > 7

```
SELECT  S.sname, S.age  
FROM    Sailors S  
WHERE   S.Rating > 7 AND
        S.age = (SELECT  MAX (S2.age)  
                   FROM    Sailors S2  
                   WHERE  S2.Rating > 7)
```
So far, we’ve applied aggregate operators to all (qualifying) tuples.

Sometimes, we want to apply them to each of several groups of tuples.

Consider: Find the age of the youngest sailor for each rating level.

- If rating values go from 1 to 10; we can write 10 queries that look like this:

For $i = 1, 2, \ldots, 10$:

```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```
GROUP BY

```
SELECT [DISTINCT] target-list
FROM relation-list
[WHERE condition]
GROUP BY grouping-list
```

Find the age of the youngest sailor for each rating level

```
SELECT S.rating, MIN(S.Age)
FROM Sailors S
GROUP BY S.rating
```
Conceptual Evaluation Strategy

- Semantics of an SQL query defined as follows:
  - Compute the cross-product of \textit{relation-list}
  - Discard resulting tuples if they fail \textit{condition}.
  - Delete attributes that are not in \textit{target-list}
  - Remaining tuples are partitioned into groups by the value of the attributes in \textit{grouping-list}
  - One answer tuple is generated per group

- Note: Does not imply query will actually be evaluated this way!
Find the age of the youngest sailor with age $\geq 18$, for each rating with at least one such sailor

$$\text{SELECT S.rating, } \text{MIN (S.age)}$$
$$\text{FROM Sailors S}$$
$$\text{WHERE S.age } \geq 18$$
$$\text{GROUP BY S.rating}$$

<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>15.5</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Answer relation
Are These Queries Correct?

SELECT MIN(S.Age)
FROM Sailors S
GROUP BY S.rating

SELECT S.name, S.rating, MIN(S.Age)
FROM Sailors S
GROUP BY S.rating
What does this query compute?

```
SELECT  B.bid,  COUNT (*) AS scount
FROM    Reserves R, Boats B
WHERE   R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```
Find those ratings for which the average age is the minimum over all ratings

```sql
SELECT Temp.rating, Temp.avgage
FROM (SELECT S.rating, AVG(S.age) AS avgage
     FROM Sailors S
     GROUP BY S.rating) AS Temp
WHERE Temp.avgage = (SELECT MIN(Temp2.avgage)
     FROM (SELECT AVG(S.age) as avgage
            FROM Sailors S
            GROUP BY S.rating) AS Temp2
     )
```
What does this query compute?

SELECT  Temp.rating, Temp.minage
FROM (SELECT  S.rating, MIN (S.age) AS minage, COUNT(*) AS cnt
   FROM  Sailors S
   WHERE  S.age >= 18
   GROUP BY  S.rating) AS Temp
WHERE  Temp.cnt >= 2
Queries With GROUP BY and HAVING

\[
\text{SELECT [DISTINCT]} \ target-list \\
\text{FROM relation-list} \\
[\text{WHERE qualification}] \\
\text{GROUP BY grouping-list} \\
\text{HAVING group-qualification}
\]

Find the age of the youngest sailor with age \( \geq 18 \) for each rating level with at least 2 such sailors

\[
\text{SELECT S.rating, MIN(S.Age)} \\
\text{FROM Sailors S} \\
\text{WHERE S.age} \geq 18 \\
\text{GROUP BY S.rating} \\
\text{HAVING COUNT(*)} \geq 2
\]
Conceptual Evaluation Strategy

• Semantics of an SQL query defined as follows:
  - Compute the cross-product of relation-list
  - Discard resulting tuples if they fail condition.
  - Delete attributes that are not in target-list
  - Remaining tuples are partitioned into groups by the value of the attributes in grouping-list
  - The group-qualification is applied to eliminate some groups
  - One answer tuple is generated per qualifying group

• Note: Does not imply query will actually be evaluated this way!
Find the age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN(S.age) FROM Sailors S WHERE S.age $\geq$ 18 GROUP BY S.rating HAVING COUNT(*) $>$ 1
```

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes `unnecessary'.
- 2nd column of result is unnamed. (Use AS to name it.)

<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>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Answer relation

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Find the age of the youngest sailor with age $\geq 18$, for each rating with at least 2 sailors (of any age)

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age $\geq 18$
GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*)
            FROM Sailors S2
            WHERE S.rating=S2.rating)
```
Find the average age for each rating, and order results in ascending order on avg. age

SELECT S.rating, AVG (S.age) AS avgage
FROM  Sailors S
GROUP BY S.rating
ORDER BY avgage

- ORDER BY can only appear in top-most query
  - Otherwise results are unordered!
Null Values

- Field values in a tuple are sometimes unknown
  - e.g., a rating has not been assigned
- Field values are sometimes inapplicable
  - e.g., no spouse's name
- SQL provides a special value null for such situations.
Queries and Null Values

SELECT S.Name 
FROM Sailors S 
WHERE S.Age > 25

• What if S.Age is NULL?
  – S.Age > 25 returns NULL!

• Implies a predicate can return 3 values
  – True, false, NULL
  – Three valued logic!

• Where clause eliminates rows that do not return true (i.e., which are false or NULL)
Three-valued Logic

SELECT S.Name
FROM Sailors S
WHERE NOT(S.Age > 25) OR S.rating > 7

• What if one or both of S.age and S.rating are NULL?

NOT Truth Table

<table>
<thead>
<tr>
<th>A</th>
<th>NOT(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

OR Truth Table

<table>
<thead>
<tr>
<th>A/B</th>
<th>True</th>
<th>False</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>True</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
**General Constraints**

- Useful when more general ICs than keys are involved
- Can use queries to express constraint
- Constraints can be named

```sql
CREATE TABLE Reserves
    ( sname CHAR(10),
    bid INTEGER, —
    day DATE,
    PRIMARY KEY (bid,day),
    CONSTRAINT noInterlakeRes
    CHECK (‘Interlake’ <>
        (SELECT B.bname
        FROM Boats B
        WHERE B.bid=bid)))
```
Constraints Over Multiple Relations

Number of boats plus number of sailors is < 100

CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100 )
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.
- SQL is the lingua franca for accessing database systems today.