Term Paper

- Due Saturday, Oct 8
- Should be about 3-4 pages (9 or 10 font)
- Some people still have not submitted topics

Homework

- Read Chapter Three
- No exercises for next class; MidTerm instead
- Any Questions?
MidTerm Exam

- Due today, September 17
- No late submissions

Homework

- Install PHP On Your System
- Install MySQL
- Create, Delete, Modify Tables
- Insert, Modify, Delete Data Into Tables
- Play with MySQL
- Any Trouble?
Oracle Buys Siebel

- September 12, 2005 – Oracle will acquire customer-service software specialist Siebel Systems in a deal worth $5.85 billion. “In a single step, Oracle becomes the No. 1 CRM applications company in the world,” said Oracle CEO Larry Ellison.

- Oracle was founded in 1977 by Larry Ellison who has a net worth of over $18 Billion, making him the 9th richest man in the world!
Relational Algebra

Chapter 4, Part A
Relational Query Languages

- Query languages (QL) - specialized languages to manipulate and retrieve data from a database
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on set theory and logic
  - Allows for much optimization
- Query Languages are programming languages!
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.
- In the summer of 1979, Relational Software, Inc. (now Oracle Corporation) introduced the first commercially available implementation of SQL (beat IBM to market by two years) by releasing their first commercial RDBMS
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 (procedural)
  - 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** (S1, S2) and **Reserves** (R1) 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>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>sname</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>sname</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 (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”)
**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**)

<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>

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

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>yuppy</td>
<td>9</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

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

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

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

\[
\pi_{\text{age}}(S2)
\]
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.
- The schema of result is identical to schema of input.
## 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`.

### Renaming operator:

\[
\rho \left( C(1 \rightarrow \text{sid1}, 5 \rightarrow \text{sid2}), S1 \times R1 \right)
\]

<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>
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>

$S1 \bowtie_{S1.sid < R1.sid} R1$

- **Result schema** same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
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>

\[ S_l \bowtie_{sid} R_l \]

- **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 $y$:

- $A/B = \left\{ \langle x \rangle | \exists \langle x, y \rangle \in A \land \forall \langle y \rangle \in B \right\}$

  - 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>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
<tr>
<td>B1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
<tr>
<td>B2</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>
<tr>
<td>B3</td>
</tr>
</tbody>
</table>

A

A/B1

A/B2

A/B3
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: \[ \rho(\text{Temp1}, \sigma_{\text{bid}=103}\text{Reserves}) \]
\[ \rho(\text{Temp2}, \text{Temp1} \bowtie \text{Sailors}) \]
\[ \pi_{\text{sname}}(\text{Temp2}) \]

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

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

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:

\[ \rho (\sigma_{\text{color} = \text{red} \lor \text{color} = \text{green}}(\text{Boats})) \]

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

- 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):

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

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

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

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

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

\[
\pi_{\text{name}} \left( \text{Tempsids} \bowtie \text{Sailors} \right)
\]
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.
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 row is essentially an assignment of constants to variables that make the formula evaluate to **true**.
Domain Relational Calculus

- **Query** has the form:
  \[ \{ \langle x_1, x_2, \ldots, x_n \rangle \mid p(\langle x_1, x_2, \ldots, x_n \rangle) \} \]

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

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

- Relational calculus is **non-operational**, and users define queries in terms of what they want, not in terms of how to compute it. *(Declarative)*
- Algebra and safe calculus have same expressive power, leading to the notion of relational completeness.
SQL: Queries, Constraints, Triggers

Chapter 5
Example Instances

- We will use these instances of the **Sailors** and **Reserves** relations in our examples.

### 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>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>

### 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>
Basic SQL Query

**SELECT**  [DISTINCT]  select-list
**FROM**  from-list
**WHERE**  qualification

- **select-list** - A list of attributes of relations in *select-list*
- **from-list** - A list of relation names (possibly with a range-variable after each name).
- **qualification** - Comparisons (Attr op const or Attr1 op Attr2, where op is one of <, >, =, \(\leq\), \(\geq\), \(\neq\)) combined using AND, OR and NOT.
- **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!
Conceptual Evaluation Strategy

Semantics of an SQL query defined in terms of the following **conceptual evaluation strategy**:

- Compute the cross-product of from-list
- Discard resulting tuples if they fail qualifications
- Delete attributes that are not in select-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
```
A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

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

OR

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

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

*It is good style, however, to use range variables always!*
Find sailors who’ve reserved at least one boat

```sql
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing `S.sid` by `S.sname` in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

\[ \pi_{\text{sname}}(\text{Sailors} \bowtie \text{Reserves}) \]
Expressions and Strings

\[
\text{SELECT } S.\text{age}, \text{age1} = S.\text{age} - 5, 2*S.\text{age} \text{ AS age2}
\text{FROM } \text{Sailors S}
\text{WHERE } S.\text{sname} \text{ LIKE 'B\_B'}
\]

- Illustrates use of arithmetic expressions and string pattern matching: 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
- \text{AS} and = are two ways to name fields in result
- \text{LIKE} is used for pattern matching. `\_` stands for any one character and `\%` stands for 0 or more arbitrary characters
- `\'Bob\'` is the only pattern match
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')
```

- If we replace **OR** by **AND** in the first version, what do we get?

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

- Same boat cannot have two colors. Always returns an **empty** answer set!
Find sid’s of sailors who’ve reserved a red or a green boat

- **UNION** - Can be used to compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries).

```
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'
```

- This query says that we want the union of the set of sailors who have reserved red boats and the set of sailors who have reserved green boats.
Find sid’s of sailors who’ve reserved a red and a green boat

◆ INTERSECT - Can be used to compute the union of any two union-compatible sets of tuples

```sql
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'
```

◆ This query has a subtle bug if we select sname instead of sid. Sname is not a key and we have two Horatio’s, each with a different color boat!
Find sid’s of sailors who’ve reserved red boats but not green boats

- **EXCEPT** - Can be used to compute set-difference of any two union-compatible sets of tuples

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='red'
EXCEPT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='green'
```
Nested Queries

Find names of sailors who’ve reserved boat #103

\[
\text{SELECT} \quad \text{S.sname} \\
\text{FROM} \quad \text{Sailors S} \\
\text{WHERE} \quad \text{S.sid IN} \quad (\text{SELECT} \quad \text{R.sid} \\
\quad \text{FROM} \quad \text{Reserves R} \\
\quad \text{WHERE} \quad \text{R.bid=103})
\]

◆ A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
◆ To find sailors who’ve not reserved #103, use **NOT IN**.
◆ To understand semantics of nested queries, think of a **nested loops** evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*
Multiply Nested Queries

Find names of sailors who have reserved a red boat

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid IN (SELECT B.bid
                                  FROM Boats B
                                  WHERE B.color = 'red'))
```
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103

\[
\text{SELECT } S.\text{sname} \\
\text{FROM } \text{Sailors } S \\
\text{WHERE } \text{EXISTS } (\text{SELECT } * \\
\text{FROM } \text{Reserves } R \\
\text{WHERE } R.\text{bid}=103 \text{ AND } S.\text{sid}=R.\text{sid})
\]

- **EXISTS** is another set comparison operator, like **IN**.
- If **UNIQUE** is used, and * is replaced by \( R.\text{bid} \), it finds sailors with at most one reservation for boat #103. (**UNIQUE** checks for duplicate tuples; * denotes all attributes)
- In general, subquery must be re-computed for each Sailors tuple.
More on Set-Comparison Operators

- We’ve already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE
- Also available: $op$ ANY, $op$ ALL, $op$ IN where $op$ is one of \{ $>$, $<$, $=$, $\geq$, $\leq$, $\neq$ \}
- Find sailors whose rating is greater than that of some sailor called Horatio

```sql
SELECT * 
FROM Sailors S 
WHERE S.rating > ANY (SELECT S2.rating 
FROM Sailors S2 
WHERE S2.sname='Horatio')
```
Rewriting INTERSECT Queries Using IN

Find sid’s of sailors who’ve reserved both a red and a green boat

```
SELECT S.sid
FROM   Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
       AND S.sid IN (SELECT S2.sid
                       FROM   Sailors S2, Boats B2, Reserves R2
                       WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                              AND  B2.color='green')
```

◆ Similarly, **EXCEPT** queries re-written using **NOT IN**

◆ To find names (not sid's) of Sailors who’ve reserved both red and green boats, just replace S.sid by S.sname in **SELECT** clause.
Division in SQL

Find sailors who’ve reserved all boats

(1)

```
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 ))
```
Division in SQL

Find sailors who’ve reserved all boats

Let’s do it the hard way, without EXCEPT:

$$\text{(2)}$$

\begin{verbatim}
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))
\end{verbatim}

Sailors S such that ...

there is no boat B without ...

a Reserves row showing S reserved B
Aggregate Operators

- Significant extension of relational algebra

Find the average age of sailors with a rating of 10

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

Find the name and age of the oldest sailor

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.age = (SELECT MAX(S2.age)
                  FROM   Sailors S2)
```

Count the number of sailors

```
SELECT COUNT (*)
FROM   Sailors S
```

Count the number of different sailor names

```
SELECT COUNT(DISTINCT S.sname)
FROM   Sailors S
```
Find name and age of the oldest sailor(s)

- The first query is illegal!
  (We’ll look into the reason a bit later, when we discuss GROUP BY)

- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = (SELECT MAX(S2.age)
            FROM Sailors S2)
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
             FROM Sailors S2 )
    = S.age
```
Motivation for Grouping

- So far, we’ve applied aggregate operators to all (qualifying) rows. Sometimes, we want to apply them to each of several groups of rows.
- Consider: Find the age of the youngest sailor for each rating level.
  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this:

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

  For \(i = 1, 2, \ldots, 10\):
Queries With GROUP BY and HAVING

- The **select-list** contains (i) **attribute names** (ii) terms with aggregate operations (e.g., MIN (S.age)).
  - The **attribute list (i)** must be a subset of **grouping-list**. Intuitively, each answer row corresponds to a **group**, and these attributes must have a single value per group. (A **group** is a set of tuples that have the same value for all attributes in **grouping-list**.)
Conceptual Evaluation

- The cross-product of from-list is computed, rows that fail qualification are discarded, `unnecessary’ fields are deleted, and the remaining rows are partitioned into groups by the value of attributes in grouping-list.
- The group-qualification is then applied to eliminate some groups. Expressions in group-qualification must have a single value per group.
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list.
- One answer row is generated per qualifying group.
Find age of the youngest sailor with age ≥18, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

**Sailors instance:**

<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>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age \( \geq 18 \), for each rating with at least 2 such sailors.

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

Sort table by groups

Apply HAVING clause

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>10</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 such sailors and with every sailor under 60.

\[
\text{HAVING COUNT (*)} > 1 \text{ AND EVERY } (S.\text{age} \leq 60)
\]

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

What is the result of changing \( \text{EVERY} \) to \( \text{ANY} \)?

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>

Introduced in SQL:1999
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 sailors between 18 and 60.

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18 AND S.age <= 60
GROUP BY S.rating
HAVING COUNT(*) > 1
```

<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>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

Answer relation:

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Null Values

◆ Field values in a row are sometimes unknown (e.g., a rating has not been assigned) or inapplicable (e.g., no spouse’s name).
  ▪ SQL provides a special value null for such situations.
◆ The presence of null complicates many issues. e.g.:
  ▪ Special operators needed to check if value is/is not null.
  ▪ Is rating>8 true or false when rating is equal to null? What about AND, OR and NOT connectives?
  ▪ We need a 3-valued logic (true, false and unknown).
  ▪ Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don’t evaluate to true.)
  ▪ New operators (in particular, outer joins) possible/needed.
Triggers

- **Trigger** - procedure that starts automatically if specified changes occur to the DBMS

- Three parts:
  - **Event** (activates the trigger)
  - **Condition** (tests whether the triggers should run)
  - **Action** (what happens if the trigger runs)
**Triggers: Example (SQL:1999)**

```sql
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON Sailors
    REFERENCING NEW TABLE NewSailors
    FOR EACH STATEMENT
    INSERT
        INTO YoungSailors(sid, name, age, rating)
    SELECT sid, name, age, rating
    FROM NewSailors N
    WHERE N.age <= 18
```
Summary

- **SQL** was an important factor in the early acceptance of the *relational model*; more natural than earlier, procedural query languages.
- **Relationally complete;** in fact, significantly **more expressive power than relational algebra**
- Even queries that can be expressed in RA can often be **expressed more naturally in SQL**
- Many *alternative ways to write a query; optimizer should look for most efficient evaluation plan.*
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.
Summary (Contd.)

- **NULL** for unknown field values brings many complications
- **Triggers** respond to changes in the database
Homework

- Read Chapters Four and Five
- Only study topics covered in class