SQL Queries

Chapter 5
The Structured Query Language

- Developed by IBM (system R) in the 1970s
- *The* most widely used language for creating, manipulating, and querying relational DBMS.
- Need for a standard since it is used by many vendors
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision)
  - SQL-99 (major extensions, current standard)
## Example Instances

- **Sailor** \((\text{sid}, \text{sname}, \text{rating}, \text{age})\)
- **Reserve** \((\text{sid}, \text{bid}, \text{day})\)
- **Boat** \((\text{bid}, \text{color})\)

<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>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>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

- relation-list A list of relation names
- target-list A list of attributes of relations in relation-list
- qualification Comparisons (Attr op const or Attr1 op Attr2, where op is one of <, >, =, ≤, ≥, ≠ ) 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!
Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:

1. Compute the cross-product of relation-list.
2. Discard resulting tuples if they fail qualifications.
3. Delete attributes that are not in target-list.
4. 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

```sql
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>
Expressions and Strings

- 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.

- AS and = are two ways to name fields in result.

- LIKE is used for string matching. _ stands for any one character and % stands for 0 or more arbitrary characters.

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM  Sailors S
WHERE  S.sname LIKE 'B_%B'
```
Find sailors who’ve reserved at least one boat

```
SELECT R.sid
FROM Reserves R
```

Would adding DISTINCT to this query make a difference?

Find the names of sailors who have reserved a red boat

```
SELECT S.name
FROM Sailors S, Reserves R, Boats B
WHERE S.sid=R.sid AND R.bid = B.bid AND B.color = 'red'
```
Find sid’s and names of sailors who’ve reserved a red or a green boat

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

- If we replace \textit{OR} by \textit{AND} in the first version, what do we get?

- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

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

```sql
SELECT S.sid, S.sname
FROM Sailors S, Boats B, Reserves R
AND B.color='red'
UNION
SELECT S.sid, S.sname
FROM Sailors S, Boats B, Reserves R
AND B.color='green'
```
Find sid’s and names of sailors who’ve reserved a red and a green boat

- **INTERSECT**: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don’t support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```sql
SELECT S.sid, S.sname
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'
```

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

SELECT S.sid, S.sname
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 and names of sailors who’ve reserved red boats but not green boats

- Sailor (sid, sname, rating, age)
- Reserve (sid, bid, day)
- Boat (bid, color)

```
SELECT S.sid, S.sname
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
EXCEPT
SELECT S.sid, S.sname
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='green'
```

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

- Many systems recognize the keyword **MINUS** instead of **EXCEPT**
Nested Queries

Find names of sailors who’ve reserved boat #103

SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
FROM Reserves R
WHERE R.bid=103)

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM 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.
Nested Queries

Find names of sailors who’ve NOT reserved boat #103

```
SELECT S.sname
FROM Sailors S
WHERE S.sid NOT IN (SELECT R.sid
FROM Reserves R
WHERE R.bid=103)
```
Nested Queries

- Sailor (sid, sname, rating, age)
- Reserve (sid, bid, day)
- Boat (bid, color)

Find names of sailors who’ve not reserved a red boat

```
SELECT S.sname
FROM Sailors S
WHERE S.sid NOT 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:

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
FROM Reserves R
WHERE R.bid=103 AND S.sid=R.sid)
```

- **EXISTS** is another set comparison operator, like **IN**.
- If **UNIQUE** is used, and * is replaced by **R.bid**, finds sailors with at most one reservation for boat #103.
- Illustrates why, 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: \( op \ \text{ANY}, \ op \ \text{ALL}, \ op \ \text{IN} \ \ >, <, =, \geq, \leq, \neq \)
- Find sailors whose rating is greater than that of some sailor called Horatio:

```
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 of sailors who’ve reserved both a red and a green boat:

```sql
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.
- Useful if your system does not support INTERSECT or EXCEPT
Division in SQL

Find sailors who’ve reserved all boats.

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

```sql
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))
```

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

- Significant extension of relational algebra.

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Bob</td>
<td>10</td>
<td>35</td>
</tr>
<tr>
<td>31</td>
<td>Bob</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

SELECT COUNT (*)
FROM Sailors S

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

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

SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'

SELECT MAX (S.age)
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.

```sql
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) 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.

  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):
    
    \[
    \text{SELECT} \ \text{MIN (S.age)} \\
    \text{FROM} \ \text{Sailors S} \\
    \text{WHERE} \ \text{S.rating} = i
    \]
    
    For \( i = 1, 2, \ldots, 10 \):

  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
Queries With GROUP BY and HAVING

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

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