Introduction to Data Management

Lecture 11
(SQL: the Saga Continues...)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- Exams and HW:
  - One exam down!
  - HW’s will now go Thursday to Thursday
- This week’s discussion session plan:
  - None! (Midterm break week.)
- Today’s lecture plan:
  - More about SQL...!
Example Data in MySQL

<table>
<thead>
<tr>
<th>Sailors</th>
<th>Reserves</th>
<th>Boats</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>bid</td>
<td>bid</td>
</tr>
<tr>
<td>22</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>29</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>103</td>
</tr>
<tr>
<td>32</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- If we replace OR by AND in this first version, what do we get?
- UNION: Can be used to compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries).
- Also available: EXCEPT (What would we get if we replaced UNION by EXCEPT?)

[Note: MySQL vs. RelaX – and why?]
SQL vs. TRC

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

\[
\{ \text{t(sid)} \mid \exists s \in \text{Sailors} \ (t.sid = s.sid \land \\
\exists r \in \text{Reserves} \ (r.sid = s.sid \land \\
\exists b \in \text{Boats} \ (b.bid = r.bid \land \\
(b.color = 'red' \lor b.color = 'green'))}) \}
\]

Find sid’s 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 not in all systems (incl. MySQL).
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

\[
\text{SELECT S.sid FROM Sailors S, Boats B, Reserves R WHERE S.sid=R.sid AND R.bid=B.bid AND (B.color='red' OR B.color='green')}
\]

\[
\text{SELECT S.sid FROM Sailors S, Boats B, Reserves R WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='green'}
\]

\[
\text{SELECT S.sid FROM Sailors S, Boats B, Reserves R WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'}
\]

\[
\]

\[
\text{SELECT S.sid FROM Sailors S, Boats B, Reserves R WHERE S.sid=R1.sid AND R1.bid=B.bid AND B.color='red'}
\]

\[
\text{SELECT S.sid FROM Sailors S, Boats B, Reserves R WHERE S.sid=R1.sid AND R1.bid=B.bid AND B.color='green'}
\]
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 and HAVING clauses!!)
- To find sailors who’ve not reserved #103, use NOT IN.
- To understand semantics (including cardinality) of nested queries, think nested loops evaluation: For each Sailors tuple, check qualification by computing subquery.

Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103:

```
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.
- Illustrates why, in general, subquery must be recomputed for each Sailors tuple (conceptually).

**NOTE:** Recall that there was a join way to express this query, too. Relational query optimizers will try to unnest queries into joins when possible to avoid nested loop query evaluation plans.
More on Set-Comparison Operators

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

\[
\text{SELECT } * \\
\text{FROM } \text{Sailors S} \\
\text{WHERE S.rating } > \text{ANY (SELECT S2.rating} \\
\text{FROM } \text{Sailors S2} \\
\text{WHERE S2.sname=’Horatio’)}
\]

Rewriting INTERSECT Queries Using IN

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

\[
\text{SELECT S.sid} \\
\text{FROM } \text{Sailors S, Boats B, Reserves R} \\
\text{WHERE S.sid=R.sid AND R.bid=B.bid AND B.color=’red’} \\
\text{AND S.sid IN (SELECT S2.sid} \\
\text{FROM Sailors S2, Boats B2, Reserves R2} \\
\text{WHERE S2.sid=R2.sid AND R2.bid=B2.bid} \\
\text{AND B2.color=’green’)}
\]

- Similarly, EXCEPT queries can be re-written using NOT IN.
- This is what you’ll need to do when using MySQL (but you can play with RelaX for the other set ops).
**Division, SQL Style**

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))  

Sailors S such that ...
the set of all Boat ids ...
minus ...
this Sailor’s reserved Boat ids...

is empty!

(2) 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))  

Sailors S such that ...
there is no boat B without ...
a Reserves tuple showing S reserved B

**Division in SQL (cont.)**

Find sailors who’ve reserved all boats.

Let’s do it the hard(er) way, i.e., without EXCEPT:

(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))  

(2) 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))
Ordering and/or Limiting Query Results

Find the ratings, ids, names, and ages of the three best sailors

```
SELECT  S.rating, S.sid, S.sname, S.age
FROM    Sailors S
ORDER BY S.rating DESC
LIMIT 3
```

- The general syntax for this:

```
SELECT [DISTINCT] expressions
FROM tables
[WHERE condition]
....
[ORDER BY expression [ ASC | DESC ]]
LIMIT number_rows [ OFFSET offset_value ];
```

Aggregate Operators

- Significant extension of the relational algebra.

```
SELECT COUNT (*)
FROM  Sailors S

SELECT AVG (S.age)
FROM  Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
    FROM  Sailors S2)

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

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

```
COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)
```
Find name and age of the oldest sailor(s)

- That first try is illegal! (We’ll see why shortly, when we do GROUP BY.)

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

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.
  - 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 (☺):

    For \( i = 1, 2, \ldots, 10 \):
    
    ```sql
    SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i
    ```
Queries With GROUP BY and HAVING

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

- The target-list contains (i) attribute names and (ii) terms with aggregate operations (e.g., MIN (S.age)).
  - The attribute list (i) must be a subset of grouping-list. Intuitively, each answer tuple 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 relation-list is computed, tuples that fail the qualification are discarded, `unnecessary’ fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- A group-qualification (HAVING) is then applied to eliminate some groups. Expressions in group-qualification must also have a single value per group!
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op must appear in grouping-list. (Note: SQL doesn’t consider primary key semantics here.)
- One answer tuple is generated per qualifying group.
Find age of the youngest sailor with age ≥ 18 for each rating with at least 2 such sailors.

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

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>

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>

Find age of the youngest sailor with age ≥ 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>

<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>
Example Data in MySQL

### Sailors

<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>45.0</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Brutus</td>
<td>33.0</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>Horatio</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>Zorba</td>
<td>16.0</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>Horatio</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>63.5</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Joan</td>
<td>35.0</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>Johan</td>
<td>35.0</td>
<td></td>
</tr>
</tbody>
</table>

### Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>1998-10-10</td>
</tr>
<tr>
<td>22</td>
<td>102</td>
<td>1998-10-10</td>
</tr>
<tr>
<td>22</td>
<td>103</td>
<td>1998-10-08</td>
</tr>
<tr>
<td>22</td>
<td>104</td>
<td>1998-10-07</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>1998-11-10</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>1998-11-06</td>
</tr>
<tr>
<td>31</td>
<td>104</td>
<td>1998-11-12</td>
</tr>
<tr>
<td>64</td>
<td>101</td>
<td>1998-09-05</td>
</tr>
<tr>
<td>64</td>
<td>102</td>
<td>1998-09-08</td>
</tr>
<tr>
<td>74</td>
<td>103</td>
<td>1998-09-08</td>
</tr>
<tr>
<td>HOLL</td>
<td>103</td>
<td>2001-01-11</td>
</tr>
<tr>
<td>HOLL</td>
<td>1</td>
<td>2002-02-02</td>
</tr>
</tbody>
</table>

### Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

For each red boat, find the number of reservations for this boat

```sql
SELECT B.bid, COUNT(*) AS scount
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

- We’re grouping over a join of three relations!
- What do we get if we remove `B.color='red'` from the WHERE clause and add a HAVING clause with this condition? (Hint: Trick question... 😊)
- What if we drop Sailors and the condition involving S.sid?
Find age of the youngest sailor with age > 18 for each rating with at least 2 sailors (of any age)

```
SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age > 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT(*)
            FROM Sailors S2
            WHERE S.rating = S2.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 or more sailors over 18!
- What if HAVING clause were replaced by:
  - HAVING COUNT(*) >1

Find those ratings for which the average age is the minimum age over all Sailors

- Aggregate operations can’t be nested! (WRONG...)

```
SELECT S.rating
FROM Sailors S
WHERE S.age = (SELECT MIN(AVG(S2.age)) FROM Sailors S2)
```

- Correct solution (in SQL/92):

```
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(age) FROM Sailors)
```

Compute the average age for each rating...

Find the overall minimum age
Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse’s name).
  - SQL provides 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. (The WHERE clause eliminates rows that don’t evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.