SQL - 4

Week 9
GROUP BY and HAVING

- 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 (!):

\[
\text{SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i}
\]

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

<table>
<thead>
<tr>
<th>SELECT</th>
<th>[DISTINCT] target-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>relation-list</td>
</tr>
<tr>
<td>WHERE</td>
<td>qualification</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>grouping-list</td>
</tr>
<tr>
<td>HAVING</td>
<td>group-qualification</td>
</tr>
</tbody>
</table>

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN \((S.\text{age})\)).
  - 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 \textit{relation-list} is computed, tuples that fail \textit{qualification} are discarded, \textquote{unnecessary} fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in \textit{grouping-list}.

• The \textit{group-qualification} is then applied to eliminate some groups. Expressions in \textit{group-qualification} must have a \textit{single value per group}!
  
  – In effect, an attribute in \textit{group-qualification} that is not an argument of an aggregate op also appears in \textit{grouping-list}. (SQL does not exploit primary key semantics here!)

• One answer tuple is generated per qualifying group.
Find the age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age >= 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.)
For each red boat, find the number of reservations for this boat

SELECT  B.bid,  COUNT (*) AS scount
FROM    Boats B, Reserves R
WHERE   R.bid=B.bid AND B.color=‘red’
GROUP BY B.bid

• Grouping over a join of two relations.
• What do we get if we remove \( B.color=‘red’ \) from the \textbf{WHERE} clause and add a \textbf{HAVING} clause with this condition?

\[
\text{SELECT  B.bid,  COUNT (*) AS scount}
\text{FROM    Boats B, Reserves R}
\text{WHERE   R.bid=B.bid}
\text{GROUP BY B.bid}
\text{HAVING B.color=‘red’}
\]

\textbf{Illegal!}

\textbf{Can be rewritten using EVERY in HAVING:}

\[
\text{SELECT  B.bid,  COUNT (*) AS scount}
\text{FROM    Boats B, Reserves R}
\text{WHERE   R.bid=B.bid}
\text{GROUP BY B.bid}
\text{HAVING EVERY(B.color=‘red’)}
\]
Find the 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 sailors over 18!
Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested!

**WRONG:**

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

**Correct solution in SQL/92 (but does not work in Oracle):**

```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 (Temp.avgage)
                     FROM Temp)
```
Continue from previous

However, this should work on Oracle 8 (or later):

```
SELECT S.rating
FROM Sailors S
GROUP BY S.rating
HAVING AVG(S.age) = (SELECT MIN (AVG (S2.age))
                      FROM Sailors S2
                      Group by rating);
```

Can use nested aggregates with Group By
Null Values

• We use *null* when the column value is either *unknown* or *inapplicable*.

• A comparison with at least one null value always returns *unknown*.

• SQL also provides a special comparison operator *IS NULL* to test whether a column value is *null*.

• To incorporate nulls in the definition of duplicates we define that two rows are duplicates if corresponding rows are equal or both contain *null*. 
Deal with the null value

• Special operators needed to check if value is/is not null.
  – “is null” always true or false (never unknown)
  – “is not null”

• Is rating > 8 true or false when rating is equal to null?
  – Actually, it’s unknown.
  – Three-valued logic
## Three valued logic

### AND

<table>
<thead>
<tr>
<th></th>
<th>False</th>
<th>True</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>False</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

### OR

<table>
<thead>
<tr>
<th></th>
<th>False</th>
<th>True</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>Unknown</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

### NOT

<table>
<thead>
<tr>
<th></th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
Other issues with the null value

- WHERE and HAVING clause eliminate rows that don’t evaluate to true (i.e., rows evaluate to false or unknown).
- Aggregate functions ignore nulls (except count (*))
- DISTINCT treats all nulls as the same
Outer Joins

• Let R and S be two tables. The outer join preserves the rows of R and S that have no matching rows according to the join condition and outputs them with nulls at the non-applicable columns.

• There exist three different variants: left outer join, right outer join and full outer join.
### Outer joins

#### (left outer-join)

<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>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>Null</td>
<td>Null</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>

= 

<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>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>Null</td>
<td>Null</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>
In Oracle

Select *
From Sailor S, Reserve R
Where S.sid = R.sid(+);

How about:

Select S.sid, count(R.bid)
From Sailor S, Reserve R
Where S.sid = R.sid(+)
Group by S.sid;

OR

Select S.sid, count(*)
From Sailor S, Reserve R
Where S.sid = R.sid(+)
Group by S.sid;
More outer joins

• Left outer join
  + sign on the right in Oracle:
  Select * from R, S where R.id=S.id(+)

• Right outer join
  + sign on the left in Oracle:
  Select * from R, S where R.id(+)=S.id

• Full outer join
  – not implemented in Oracle 8
  – Added for Oracle 9 (or later)
    • Use full text instead of +’s: “full outer join”, “left outer join”, “right outer join”, “inner join”
Value functions

• Values can be transformed before aggregated:
  Select sum(S.A/2) from S;

• An interesting decode function (Oracle specific):
  decode(value, if1, then1, if2, then2, …, else):

  Select sum(decode(major, ‘CS’, 1, 0)) as Num_CS_Stu,
  sum(decode(major, ‘CS’, 0, 1)) as Num_NonCS_Stu
  From student;

  if (major == ‘CS’)
    result = 1;
  else
    result = 0;
Overall:
Conceptual order in query evaluation

- First the relational products of the tables in the `FROM` clause are evaluated.
- From this, rows not satisfying the `WHERE` clause are eliminated.
- The remaining rows are grouped in accordance with the `GROUP BY` clause.
- Groups not satisfying the `HAVING` clause are then eliminated.
- The expressions in the `SELECT` list are evaluated.
- If the keyword `DISTINCT` is present, duplicate rows are now eliminated.
- Evaluate `UNION`, `INTERSECT` and `EXCEPT` for Subqueries up to this point.
- Finally, the set of all selected rows is sorted if the `ORDER BY` is present.
Conclusion

• Nested queries are a very powerful feature in SQL; they help us write shorter and more efficient queries.
• Post processing on the result of queries is supported.
• Aggregation is the most complex “post processing”
  – “Group by” clause partition the results into groups
  – “Having” clause puts condition on groups (just like Where clause on tuples).
Exercises on your social network database

• Print the number of opposite-sexed friends each user X has, for each user with at least 10 such friends. Again, assume asymmetric friendship and consider only people X befriends with.

• Print the number of opposite-sexed friends each user X has, for each user with at least 10 friends. Again, assume asymmetric friendship and consider only people X befriends with.