SQL: Having

Filtering groups with HAVING

We've had lots of experience using a `WHERE` clause to filter individual tuples. For example

```
csc343h-prof=> SELECT *
csc343h-prof-> FROM Offering
csc343h-prof-> WHERE dept = 'CSC';
```

In the result table, we get exactly those rows that satisfy the `WHERE` condition. Similarly, we can filter *groups* using a `HAVING` clause. For example, suppose we want to know the average grade and the number of students in each offering of any course:

```
csc343h-prof=> SELECT oid, avg(grade), count(*)
csc343h-prof-> FROM Took
csc343h-prof-> GROUP BY oid;
 oid |         avg         | count
-----+---------------------+-------
  14 | 59.0000000000000000 |     3
  34 | 60.6666666666666667 |     3
  27 | 70.5000000000000000 |     2
   8 | 92.0000000000000000 |     2
  17 | 69.5000000000000000 |     4
  28 | 91.0000000000000000 |     1
   1 | 87.2500000000000000 |     4
  15 | 31.0000000000000000 |     2
  26 | 71.0000000000000000 |     1
  11 | 79.0000000000000000 |     4
  38 | 92.0000000000000000 |     1
  16 | 73.5000000000000000 |     4
  39 | 97.0000000000000000 |     1
   6 | 74.0000000000000000 |     3
  21 | 71.0000000000000000 |     1
   3 | 82.0000000000000000 |     1
  31 | 78.0000000000000000 |     4
  35 | 75.0000000000000000 |     1
   5 | 74.6666666666666667 |     3
  13 | 95.6666666666666667 |     3
  22 | 75.0000000000000000 |     2
   9 | 78.0000000000000000 |     1
   7 | 83.0000000000000000 |     3
(23 rows)
```

We might find the average interesting only in offerings with multiple students. We can filter out the other offerings with a `HAVING` clause:

```
csc343h-prof=> SELECT oid, avg(grade), count(*)
csc343h-prof-> FROM Took
csc343h-prof-> GROUP BY oid
csc343h-prof-> HAVING count(*) > 1;
 oid |         avg         | count
-----+---------------------+-------
  14 | 59.0000000000000000 |     3
  34 | 60.6666666666666667 |     3
  27 | 70.5000000000000000 |     2
   8 | 92.0000000000000000 |     2
  17 | 69.5000000000000000 |     4
   1 | 87.2500000000000000 |     4
  15 | 31.0000000000000000 |     2
  11 | 79.0000000000000000 |     4
  16 | 73.5000000000000000 |     4
   6 | 74.0000000000000000 |     3
  31 | 78.0000000000000000 |     4
   5 | 74.6666666666666667 |     3
  13 | 95.6666666666666667 |     3
  22 | 75.0000000000000000 |     2
   7 | 83.0000000000000000 |     3
(15 rows)
```

We can also filter according to the value of an unaggregated attribute:

```
csc343h-prof=> SELECT oid, avg(grade), count(*)
csc343h-prof-> FROM Took
csc343h-prof-> GROUP BY oid
csc343h-prof-> HAVING oid > 10;
 oid |         avg         | count
-----+---------------------+-------
  14 | 59.0000000000000000 |     3
  34 | 60.6666666666666667 |     3
  27 | 70.5000000000000000 |     2
  17 | 69.5000000000000000 |     4
  28 | 91.0000000000000000 |     1
  15 | 31.0000000000000000 |     2
  26 | 71.0000000000000000 |     1
  11 | 79.0000000000000000 |     4
  38 | 92.0000000000000000 |     1
  16 | 73.5000000000000000 |     4
  39 | 97.0000000000000000 |     1
  21 | 71.0000000000000000 |     1
  31 | 78.0000000000000000 |     4
  35 | 75.0000000000000000 |     1
  13 | 95.6666666666666667 |     3
  22 | 75.0000000000000000 |     2
```

```
    (16 rows)
```

Restrictions on HAVING clauses

A `HAVING` clause may refer to an attribute only if it is either aggregated or is an attribute on the GROUP BY list. This is the same requirement as for SELECT clauses with aggregation, and it makes sense for the same reason. It's all to do with quantities.

Here's an example. It's the same query we just saw, but this time we filter using grade unaggregated

```
    csc343h-prof=> SELECT oid, avg(grade), count(*)
    csc343h-prof-> FROM Took
    csc343h-prof-> GROUP BY oid
    csc343h-prof-> HAVING grade < 50;
    ERROR:  column "took.grade" must appear in the GROUP BY clause or be used in an aggregate functi
on
    LINE 4: HAVING grade < 50;
                   ^
```

It makes complete sense that this doesn't work. We have grouped by oid, so there will be one row per oid (for oids that pass the filter). But the filter is `grade < 50`, and there is not just one grade to check per oid.

If we change the filter to min(grade) < 50, the query makes sense again. There is only one minimum grade per oid, so we can compare it to 50.

```
    csc343h-prof=> -- We include a column for min(grade) so we can observe
    csc343h-prof=> -- that the filter is working.
    csc343h-prof=> SELECT oid, min(grade), avg(grade), count(*)
    csc343h-prof-> FROM Took
    csc343h-prof-> GROUP BY oid
    csc343h-prof-> HAVING min(grade) < 50;
     oid | min |         avg         | count
    -----+-----+---------------------+-------
      14 |  39 | 59.0000000000000000 |     3
      34 |  45 | 60.6666666666666667 |     3
      17 |  46 | 69.5000000000000000 |     4
      15 |   0 | 31.0000000000000000 |     2
      11 |  39 | 79.0000000000000000 |     4
      16 |  17 | 73.5000000000000000 |     4
    (6 rows)
```

We can even filter on something that is not in the `SELECT` clause. Here we have the same query, except that we have taken `min(grade)` out of the `SELECT` clause so that it is not reported in the result. The query still works.

```
csc343h-prof=> SELECT oid, avg(grade), count(*)
csc343h-prof-> FROM Took
csc343h-prof-> GROUP BY oid
csc343h-prof-> HAVING min(grade) < 50;
 oid |         avg         | count
-----+---------------------+-------
  14 | 59.0000000000000000 |     3
  34 | 60.6666666666666667 |     3
  17 | 69.5000000000000000 |     4
  15 | 31.0000000000000000 |     2
  11 | 79.0000000000000000 |     4
  16 | 73.5000000000000000 |     4
(6 rows)
```

Note that the `HAVING` clause is executed *before* the `SELECT` clause, so the `SELECT` has not yet filtered out the `grade` column.

Order of execution:

Now that we know all the possible clauses in a SELECT-FROM-WHERE query, we can consider the order of execution. It is as follows:

1. FROM clause: This determines what table(s) will be examined.
   - If there is more than one, the tables are joined together as specified. In any case, the rows are iterated over.
2. WHERE clause: This filters rows.
   - Notice that the SELECT has not happened yet, so we can't reference any column names that it defines.
3. GROUP BY clause: This organizes the rows into groups, each of which will be represented by one row in the result table.
   - If there is also a HAVING clause, only groups that pass its filter are included.
4. HAVING clause: This filters groups.
   - Since this happens before the SELECT clause, it can refer to attributes that are not included in the SELECT.
5. SELECT clause: This chooses which columns to include in the result.
   - It may introduce new column names.
6. ORDER BY clause: This sorts the rows of the result table.
   - Since it occurs after the SELECT clause, it can reference column names that are introduced there.