SQL 3

In this lab we will introduce R's Postgres client, *RPostgreSQL*. We'll use it to round out our coverage of SQL by exploring `HAVING` and `JOIN` commands.

R's RPostgreSQL

The documentation for *RPostgreSQL* can be found online. You can install it in the normal way with `install.packages`.

The `HAVING` Command

Thus far, we have learned various commands for selecting, filtering, and extracting data. The most recent thing we learned was `GROUP BY`. Let's now introduce `HAVING`, which allows you to filter (much like a `WHERE`) on aggregate functions *after* the grouping has occurred:

```
SELECT
FROM
WHERE
GROUP BY
*HAVING*
ORDER BY
LIMIT
```

To demonstrate with an example, say we wanted to get a count of each race:

```
SQL = """
SELECT race, COUNT(race) AS population
FROM cmspop
GROUP BY race
ORDER BY race ASC;
"""
tabular(SQL)
```

```
       race  population
0     black      240294
1  hispanic       52799
2    others       95012
3     white     1866993
```

This should be an easy query for you to make at this point (after having completed SQL HW 2). As a side note, in Python you can create blocks of strings that span multiple lines by leading and ending the string with triple quotes, like above.

To get races where the count is above 100,000 you would normally utilize a nested query:

```
SQL = """
SELECT * FROM
    (SELECT race, COUNT(race) AS population
    FROM cmspop
    GROUP BY race
    ORDER BY race ASC) AS sub_query
WHERE population > 100000;
"""
tabular(SQL)
```

```
    race  population
0  black      240294
1  white     1866993
```

But there's actually another way to filter these results: by using `HAVING`. In fact, that is specifically what `HAVING` does. It filters the results of aggregate functions (`GROUP BY`) **after** the aggregation has been performed. We can get the same results as the above query by eliminating the nested query and using `HAVING`, instead:

```
SQL = """
SELECT race, COUNT(race) AS population
FROM cmspop
GROUP BY race
HAVING COUNT(race) > 100000
ORDER BY race ASC;
"""
tabular(SQL)
```

```
    race  population
0  black      240294
1  white     1866993
```

It's important to remember that `HAVING` is run after the aggregation because this will impact performance. If, for example, you wanted to remove the race `others` from your results, you should do it with a `WHERE` command rather than with `HAVING` since `WHERE` will remove all the `race=others` **before** the aggreation, saving computation.

We won't cover `HAVING` in much detail because it's clear from the above that there are ways to operate without ever using `HAVING`. However, it's a convenient tool to have when you want to quickly filter the results of an aggregation.

`JOIN` Commands

The remainder of our time with SQL will be spent with `JOIN`s. We will cover five types of joins.

Left Join

Joins everything highlighted in table A with the highlight in table B. They have to be similar data types (e.g. you can match chars and varchars; ints and doubles).

```
____A____
|       |
|       |
|       |
|       |
              __B__
             |     |
             |     |
             |     |
             -------
```

Right Join

Just the opposite...takes all of table B and just the "overlapping" (i.e. common) part of table A.

Inner Join

Only joins the information in both tables.

```
____A____
|       |
|       |
|       |
|       |     __B__
|       |    |     |
             |     |
             |     |
             -------
```

Outer Join

Joins everything. This can be thought of as a union of A and B (all the rows in A and B). If one table doesn't have a matching data point in the other column, then a NULL will be created.

```
____A____
|       |
|       |
|       |
|       |
              __B__
             |     |
             |     |
             |     |
             -------
```

Cross Join

This is a cross-product of all rows and columns in both tables. If you have two 5x5 tables and you cross join them, you'll end up with a 5 column by 25 row table.

`JOIN` Syntax

A `JOIN` is made up of several important components:

1. The table A
2. The table B
3. The type of join being made
4. The columns to join on

These components are put together like this:

```
SELECT ....
FROM
    table A
LEFT JOIN
    table B
ON A.race = B.race AND A.sex = B.sex;
```

Above, we've created a psuedo-statement to `LEFT JOIN` table A with table B where the `race` in A is the same as the `race` in B and the `sex` in A is the same as the `sex` in B.

`table A` can be any table: it can be a table in your database, or it can be a table returned by another SQL query. If the latter, you should name what is returned, just liked you name sub-queries. We'll see this in examples in a moment.

In our tables `cmspop` and `cmsclaims` the column `id` is common to both tables. Using this, we can join the two tables together, collecting the claims data for each row next to the data about the patient:

```
SQL = """
SELECT cmspop.id, cmspop.dob, cmspop.sex, cmspop.state, cmsclaims.hmo_mo,
cmsclaims.carrier_reimb
FROM
    cmspop
INNER JOIN
    cmsclaims
ON cmspop.id = cmsclaims.id
LIMIT 10;
"""
tabular(SQL)
```

```
                 id         dob     sex state  hmo_mo  carrier_reimb
0  001E248F6DB5B893  1967-02-01  female    CA       0             30
1  001EA2F4DB30F105  1925-07-01    male    IL       0           1820
2  002A425E967ED186  1941-04-01    male    FL      12           1110
```

```
3  0036004F5BAF9171  1913-02-01  female    AZ       0            1220
4  0067BBCE45146AF6  1933-01-01  female    TX       0            2070
5  007B0277AB60C3B0  1940-12-01    male    MA       0            1400
6  007F679BBEE4E890  1941-01-01    male    WA      12             860
7  009ED6EC0FDB2E23  1927-12-01    male    CA      12              30
8  00A81AC19FA0F186  1930-10-01    male    FL       0            1470
9  00B7FD9325DDC843  1942-09-01    male    FL       0            1480
```

Aha! Finally, columns from both our tables in the same place! Notice how on the first line of the SQL statement we had to specify which table each column comes from using dot notation. This isn't always necessary, but quite often is, so pay attention for this if you get any errors.

We can collect the results of our `JOIN` and then use `GROUP BY` on it, as well:

```
SQL = """
SELECT cmspop.sex, AVG(cmsclaims.carrier_reimb)
FROM
    cmspop
INNER JOIN
    cmsclaims
ON cmspop.id = cmsclaims.id
GROUP BY cmspop.sex
LIMIT 10;
"""
tabular(SQL)
```

```
      sex                   avg
0    male  815.4688044197439447
1  female  873.8136098428712684
```

Now, let's say we wanted to join only by the state CA. In this case, we should first filter our table `cmspop` to get only that state, **then** make the join. Joins are expensive operations, so if we can limit what we're joining that is always worthwhile.

```
SQL = """
SELECT LHS.sex, AVG(RHS.carrier_reimb)
FROM
    (SELECT * FROM cmspop WHERE state='CA') AS LHS
LEFT JOIN
    (SELECT * FROM cmsclaims) AS RHS
ON LHS.id = RHS.id
GROUP BY LHS.sex
LIMIT 10;
"""
tabular(SQL)
```

```
      sex                   avg
0  female  791.9395441119699785
1    male  786.1829746760194983
```

You should read the above query carefully. Notice how the tables `cmspop` and `cmsclaims` have been replaced with `SELECT` statements; what those select statements return have also been named `LHS` and `RHS`. These stand for Left-Hand Side and Right-Hand Side. They represent the left and right of your join (table A and table B). You should get in the habit of naming the tables in your joins `LHS` and `RHS`. If there comes a time where another name is more explanatory, it's ok to use that. Just always make sure the names you use for nested queries and in joins is explanatory.

Now it's your turn. Complete the questions below. Just like on the homework, you should create a single SQL query, return only what you are asked, and only use what has been taught in class to this point. You are free to collaborate with your neighbors. You can also test your queries in *psql*, but once you have the right query it should be used here to produce an output with *psycopg2* using the `tabular()` function defined above.

Q1

Find the average months of HMO coverage when the patient was reported to have cancer.

Q2

Return the top five rows where the age of death is below the average age of death, ordered by `id` in ascending order.

Q3

Find the total carrier reimbursements for every state, ordered by state in ascending order.

Q4

Find out which state spends the most money on carrier reimbursements for depression. Return columns for the `state` and total carrier reimbursements spent on depression.

Q5

Rank each state by their number of heart failure claims *in proportion to their total claims*, ordered by the proportion in descending order. Your query should return two columns.

Q6

For everyone who is deceased, find their deviation in age from the average age of the deceased, in years and rounded to two decimal places. Return columns for `id`, `sex`, `race`, and `deviation` from the average age.

Q7

Imagine you are doing a social study on the health of certain races in different regions of the country. For the race that most frequently submits claims in Texas, find the state which has the lowest frequency of claims from the same race. Return the percentage of carrier reimbursement cost that race is responsible for, the average number of HMO months of coverage, and average beneficiary responsibility for those two states and that race. Order by state in ascending order.

Q8

Imagine you are an insurance company and you want to know which ailments are the most common, when they usually occur, and how much they cost. Return the percentage of claims to the nearest two decimal places for each ailment (Alzheimers, heart failure, etc.) along with the average age of patients *in integer years* and the average carrier reimbursement for those ailments.