LEDE PROGRAM: DATA AND DATABASES DAY 4

Here's my summary yesterday's SQL fun in class.

We talked about JOIN with a nice Venn diagram from this Stack overflow page: https://stackoverflow.com/questions/13997365/sql-joins-as-venn-diagram

And we did a bunch of different joins on the very tiny friends table from the first day of class. (We noted that there was an extra table in there that has been automatically created because of the sequence type that I had added.)

```
dayone=# \d
               List of relations
 Schema |         Name          |   Type   | Owner
--------+-----------------------+----------+-------
 public | friends               | table    | Jon
 public | friends_friend_id_seq | sequence | Jon
 public | messages              | table    | Jon
(3 rows)

dayone=# \d friends_friend_id_seq
             Sequence "public.friends_friend_id_seq"
  Type   | Start | Minimum |  Maximum   | Increment | Cycles? | Cache
---------+-------+---------+------------+-----------+---------+-------
 integer |     1 |       1 | 2147483647 |         1 | no      |     1
Owned by: public.friends.friend_id

dayone=# SELECT * FROM friends_friend_id_seq ;
 last_value | log_cnt | is_called
------------+---------+-----------
          7 |      32 | t
(1 row)

dayone=# SELECT * FROM friends;
 age |    name     |  birthday  | friend_id | still_friend
-----+-------------+------------+-----------+--------------
  52 | Keanu       | 1964-09-02 |         1 | t
  52 | Keanu       | 1964-09-02 |         2 | t
  52 | Keanu       | 1964-09-02 |         3 | t
  49 | Carrie-Anne | 1967-08-21 |         5 | t
  57 | Hugo        | 1960-04-04 |         6 | t
  33 | Sam         | 1986-02-07 |         7 | t
(6 rows)

dayone=# SELECT * FROM messages;
 friend_id |            time            |         message
-----------+----------------------------+--------------------------
         1 | 2019-05-29 12:53:30.57085  | YO
         1 | 2019-05-29 12:53:30.57085  | HEY
         1 | 2019-05-29 12:53:30.57085  | how are you doing today?
         7 | 2019-05-29 13:02:29.020568 | Hello!
(4 rows)
```

We started with an INNER JOIN which is the same thing as JOIN. That finds common columns and returns the least number of rows having common keys. There are four rows in messages, and they join with one row each from friends. 4 X 1 = 4

```
dayone=# SELECT * FROM messages JOIN friends ON friends.friend_id = messages.friend_id;
 friend_id |            time            |         message          | age | name  |  birthday  | friend_id | still_friend
-----------+----------------------------+--------------------------+-----+-------+------------+-----------+--------------
         1 | 2019-05-29 12:53:30.57085  | YO                       |  52 | Keanu | 1964-09-02 |         1 | t
         1 | 2019-05-29 12:53:30.57085  | HEY                      |  52 | Keanu | 1964-09-02 |         1 | t
         1 | 2019-05-29 12:53:30.57085  | how are you doing today? |  52 | Keanu | 1964-09-02 |         1 | t
         7 | 2019-05-29 13:02:29.020568 | Hello!                   |  33 | Sam   | 1986-02-07 |         7 | t
(4 rows)
```

Next we did a LEFT JOIN of 'messages' with 'friends'. Since there are four rows in 'messages'. This looks pretty similar to the last join, even though the logic is different. A left join keeps the first table stable and adds columns from the second table to it.

```
dayone=# SELECT * FROM messages LEFT JOIN friends ON friends.friend_id = messages.friend_id;
 friend_id |            time            |         message          | age | name  |  birthday  | friend_id | still_friend
-----------+----------------------------+--------------------------+-----+-------+------------+-----------+--------------
         1 | 2019-05-29 12:53:30.57085  | YO                       |  52 | Keanu | 1964-09-02 |         1 | t
         1 | 2019-05-29 12:53:30.57085  | HEY                      |  52 | Keanu | 1964-09-02 |         1 | t
         1 | 2019-05-29 12:53:30.57085  | how are you doing today? |  52 | Keanu | 1964-09-02 |         1 | t
         7 | 2019-05-29 13:02:29.020568 | Hello!                   |  33 | Sam   | 1986-02-07 |         7 | t
(4 rows)
```

So if we flip which table comes first, for the LEFT JOIN , you will see the logic reveal itself. 8 rows, because it keeps all the rows from 'friends' but also adds the new rows from 'messages'. Think of it this way: in the inner join, the tables are being brought together. In the left join one table is being joined to the other table.

```
age |    name     |  birthday  | friend_id | still_friend | friend_id |            time            |         message
-----+-------------+------------+-----------+--------------+-----------+----------------------------+--------------------------
  52 | Keanu       | 1964-09-02 |         1 | t            |         1 | 2019-05-29 12:53:30.57085  | YO
  52 | Keanu       | 1964-09-02 |         1 | t            |         1 | 2019-05-29 12:53:30.57085  | HEY
  52 | Keanu       | 1964-09-02 |         1 | t            |         1 | 2019-05-29 12:53:30.57085  | how are you doing today?
  33 | Sam         | 1986-02-07 |         7 | t            |         7 | 2019-05-29 13:02:29.020568 | Hello!
  52 | Keanu       | 1964-09-02 |         2 | t            |           |                            |
  49 | Carrie-Anne | 1967-08-21 |         5 | t            |           |                            |
  57 | Hugo        | 1960-04-04 |         6 | t            |           |                            |
  52 | Keanu       | 1964-09-02 |         3 | t            |           |                            |
```

```
(8 rows)

(there are two added rows, because friend_id 1 matches three times, so there was once only one row in friends for friend_id 1, now there are two extra rows
dayone=# SELECT * FROM friends JOIN messages ON friends.friend_id = messages.friend_id;
 age | name  |  birthday  | friend_id | still_friend | friend_id |            time            |         message
-----+-------+------------+-----------+--------------+-----------+----------------------------+-------------------------
  52 | Keanu | 1964-09-02 |         1 | t            |         1 | 2019-05-29 12:53:30.57085  | YO
  52 | Keanu | 1964-09-02 |         1 | t            |         1 | 2019-05-29 12:53:30.57085  | HEY
  52 | Keanu | 1964-09-02 |         1 | t            |         1 | 2019-05-29 12:53:30.57085  | how are you doing today?
  33 | Sam   | 1986-02-07 |         7 | t            |         7 | 2019-05-29 13:02:29.020568 | Hello!
(4 rows)

(simplified columns for the same join)
dayone=# SELECT friends.name the_name, friends.friend_id the_id, messages.message  FROM friends LEFT JOIN messages ON friends.friend_id = messages.friend_i
  the_name   | the_id |         message
-------------+--------+-------------------------
 Keanu       |      1 | YO
 Keanu       |      1 | HEY
 Keanu       |      1 | how are you doing today?
 Sam         |      7 | Hello!
 Keanu       |      2 |
 Carrie-Anne |      5 |
 Hugo        |      6 |
 Keanu       |      3 |
(8 rows)
```

Next, we did a CROSS JOIN which creates every possible combination of rows from the two tables. 6 x 4 = 24

```
dayone=# SELECT friends.name the_name, friends.friend_id the_id, messages.message  FROM friends CROSS JOIN messages;
  the_name   | the_id |         message
-------------+--------+-------------------------
 Keanu       |      1 | YO
 Keanu       |      2 | YO
 Keanu       |      3 | YO
 Carrie-Anne |      5 | YO
 Hugo        |      6 | YO
 Sam         |      7 | YO
 Keanu       |      1 | HEY
 Keanu       |      2 | HEY
 Keanu       |      3 | HEY
 Carrie-Anne |      5 | HEY
 Hugo        |      6 | HEY
 Sam         |      7 | HEY
 Keanu       |      1 | how are you doing today?
 Keanu       |      2 | how are you doing today?
 Keanu       |      3 | how are you doing today?
 Carrie-Anne |      5 | how are you doing today?
 Hugo        |      6 | how are you doing today?
 Sam         |      7 | how are you doing today?
 Keanu       |      1 | Hello!
 Keanu       |      2 | Hello!
 Keanu       |      3 | Hello!
 Carrie-Anne |      5 | Hello!
 Hugo        |      6 | Hello!
 Sam         |      7 | Hello!
(24 rows)
```

Next, we moved to the UN tables. Here's a link to the instructions be used. We created a database, made tables, and tried to copy the CSV file.

```
dayone=# create database unenergy2019;
CREATE DATABASE
dayone=# \c unenergy2019
You are now connected to database "unenergy2019" as user "Jon".
unenergy2019=# \d
Did not find any relations.
unenergy2019=# CREATE TABLE solar (
unenergy2019(#   country varchar(80),
unenergy2019(#   type varchar(80),
unenergy2019(#   year int,
unenergy2019(#   unit varchar(80),
unenergy2019(#   usage double precision,
unenergy2019(#   notes varchar(80)
unenergy2019(# );
CREATE TABLE
unenergy2019=# \d solar
                    Table "public.solar"
 Column  |         Type          | Collation | Nullable | Default
---------+-----------------------+-----------+----------+---------
 country | character varying(80) |           |          |
 type    | character varying(80) |           |          |
 year    | integer               |           |          |
 unit    | character varying(80) |           |          |
 usage   | double precision      |           |          |
 notes   | character varying(80) |           |          |

unenergy2019=# \copy solar from /Users/Jon/Documents/Columbia2019/import_csv/UNdata_Solar.csv delimiter ',' csv
COPY 1270
```

We began exploring the solar table. They're duplicates of countries because there are multiple years.

```
unenergy2019=# select country from solar order by country;
          country
----------------------------
 Algeria
 Algeria
```

```
Algeria
American Samoa
American Samoa
American Samoa
American Samoa
American Samoa
Antigua and Barbuda
Antigua and Barbuda
Argentina
Argentina
Argentina
Argentina
Argentina
Argentina
Argentina
Argentina
Armenia
Armenia
Armenia
Australia
Australia
Australia
Australia
Australia
Australia
Australia
Australia
Australia
Australia
```

We used DISTINCT to see what countries are in there and if there is any misspellings (same country but wrong spelling).

```
unenergy2019=# select distinct country from solar order by country;
           country
-----------------------------
 Algeria
 American Samoa
 Antigua and Barbuda
 Argentina
 Armenia
 Australia
 Austria
 Azerbaijan
 Bahrain
 Bangladesh
 Belarus
 Belgium
 Benin
 Bolivia (Plur. State of)
 Bosnia and Herzegovina
 Botswana
 Brazil
 British Virgin Islands
 Brunei Darussalam
 Bulgaria
 Cabo Verde
 Cambodia
 Canada
 Chile
 China
 Cook Islands
 Costa Rica
 Croatia
 CuraÃ§ao
 Cyprus
 Czechia
 Denmark
 Dominican Republic
 Ecuador
 Egypt
 Eritrea
 Finland
 France
 French Guiana
 French Polynesia
 Gabon
 Germany
 Ghana
 Greece
 Guadeloupe
 Guam
 Guatemala
 Honduras
 Hungary
 India
 Indonesia
 Iran (Islamic Rep. of)
 Ireland
 Israel
 Italy
 Jamaica
 Japan
 Jordan
 Kazakhstan
 Kiribati
 Korea, Republic of
```

```
 Kuwait
 Lao People's Dem. Rep.
 Libya
 Liechtenstein
 Lithuania
 Luxembourg
 Madagascar
 Malaysia
 Maldives
 Mali
 Malta
 Marshall Islands
 Martinique
 Mauritania
 Mauritius
```

And then looked at DISTINCT countries and years as a prelude to seeing if we have any duplicates of countries and years. We don't want to have multiple rows for the same country and year--which I purposely added into make this a difficult database.

```
 unenergy2019=# select distinct country, year from solar order by country, year;
           country           | year
-----------------------------+------
 Algeria                     | 2015
 Algeria                     | 2016
 Algeria                     | 2017
 American Samoa              | 2012
 American Samoa              | 2013
 American Samoa              | 2014
 American Samoa              | 2015
 American Samoa              | 2016
 Antigua and Barbuda         | 2010
 Antigua and Barbuda         | 2011
 Argentina                   | 2009
 Argentina                   | 2010
 Argentina                   | 2011
 Argentina                   | 2012
 Argentina                   | 2013
 Argentina                   | 2014
 Argentina                   | 2015
 Argentina                   | 2016
 Armenia                     | 2015
 Armenia                     | 2016
 Armenia                     | 2017
 Australia                   | 1993
 Australia                   | 1994
 Australia                   | 1995
 Australia                   | 1996
 Australia                   | 1997
 Australia                   | 1998
 Australia                   | 1999
 Australia                   | 2000
 Australia                   | 2001
 Australia                   | 2002
 Australia                   | 2003
 Australia                   | 2004
 Australia                   | 2005
 Australia                   | 2006
 Australia                   | 2007
 Australia                   | 2008
 Australia                   | 2009
 Australia                   | 2010
 Australia                   | 2011
 Australia                   | 2012
 Australia                   | 2013
 Australia                   | 2014
 Australia                   | 2015
 Australia                   | 2016
 Austria                     | 1993
 Austria                     | 1994
 Austria                     | 1995
 Austria                     | 1996
```

I then counted all of the rows in the solar table. And all of the rows in the sub-table that had the distinct country and year. If there were no duplicates these counts should match.

```
 unenergy2019=# select count(*) from solar;
 count
-------
  1270
(1 row)

unenergy2019=# select count(*) from (select distinct country, year from solar order by country, year) as sub;
 count
-------
  1268
(1 row)
```

They didn't! Because of the two duplicates we added. So, to get more specific, I decided to count things up using GROUP BY but grouping by two columns -- country AND year!

```
unenergy2019=# SELECT COUNT(country), country, year FROM solar group by country, year ORDER BY COUNT(country) DESC;
 count |           country           | year
```

```
-------+----------------------------+------
     2 | Australia                  | 2016
     2 | Australia                  | 2011
     1 | Austria                    | 2009
     1 | Mayotte                    | 2014
     1 | United States              | 2007
     1 | Portugal                   | 2015
     1 | Costa Rica                 | 2013
     1 | Hungary                    | 2013
     1 | Cabo Verde                 | 2007
     1 | American Samoa             | 2014
     1 | Switzerland                | 2010
     1 | Romania                    | 2007
     1 | Hungary                    | 2012
     1 | Mauritius                  | 2014
     1 | Gabon                      | 2015
     1 | United States              | 1999
     1 | Belgium                    | 2005
     1 | Mayotte                    | 2015
     1 | Korea, Republic of         | 1991
     1 | Venezuela (Bolivar. Rep.)  | 2015
     1 | Switzerland                | 2016
     1 | St. Helena and Depend.     | 2013
     1 | Austria                    | 2006
     1 | Spain                      | 2005
     1 | Indonesia                  | 2016
     1 | Bolivia (Plur. State of)   | 2012
     1 | United Rep. of Tanzania    | 2013
     1 | India                      | 2013
     1 | Korea, Republic of         | 2014
     1 | Romania                    | 2009
     1 | Cook Islands               | 2011
```

Then I looked specifically at those countries and years to see what the rows looked like.

```
 unenergy2019=# SELECT country, year, usage FROM solar WHERE country = 'Australia' AND year = 2016;
  country  | year | usage
-----------+------+-------
 Australia | 2016 |  6209
 Australia | 2016 |  6209
(2 rows)

unenergy2019=# SELECT country, year, usage FROM solar WHERE country = 'Australia' AND year = 2011;
  country  | year | usage
-----------+------+-------
 Australia | 2011 |  1391
 Australia | 2011 |    13
(2 rows)
```

One set of duplicates is completely identical. The other one has a discrepancy in usage. So I queried that country to see which usage seems like the right value.

```
unenergy2019=# SELECT country, year, usage FROM solar WHERE country = 'Australia' ORDER BY year;
  country  | year | usage
-----------+------+-------
 Australia | 1993 |    11
 Australia | 1994 |    13
 Australia | 1995 |    16
 Australia | 1996 |    19
 Australia | 1997 |    23
 Australia | 1998 |    28
 Australia | 1999 |    34
 Australia | 2000 |    38
 Australia | 2001 |    44
 Australia | 2002 |    50
 Australia | 2003 |    59
 Australia | 2004 |    69
 Australia | 2005 |    79
 Australia | 2006 |    91
 Australia | 2007 |   109
 Australia | 2008 |   127
 Australia | 2009 |   160
 Australia | 2010 |   389
 Australia | 2011 |  1391
 Australia | 2011 |    13
 Australia | 2012 |  2325
 Australia | 2013 |  3475
 Australia | 2014 |  4010
 Australia | 2015 |  5023
 Australia | 2016 |  6209
 Australia | 2016 |  6209
(26 rows)
```

Looking at the steady rising values the 13 seems to be obviously a typo (seems!). So we deleted the row that had 13 in it. I almost deleted both rows with 13, but was saved at the last minute, and added year as well.

```
unenergy2019=# DELETE from solar where country = 'Australia' AND usage = 13 AND year = 2011;
DELETE 1
```

The other set of duplicates is exactly the same. This poses a bigger challenge, because we have no criteria to delete one of them. In a well-maintained database this shouldn't happen, but there are hidden differences that we can use. ctid is an identifier Postgres uses to keep track of individual rows. While we can't directly access and edit it, we can filter out the duplicate row using the following delete statement.

```
unenergy2019=# SELECT ctid FROM solar where country = 'Australia' AND year = 2016;
  ctid
```

```
--------
 (0,22)
 (0,23)
(2 rows)

 DELETE FROM solar
 WHERE ctid NOT IN
 (SELECT MAX(dt.ctid)
         FROM   Solar As dt
         GROUP BY country, year, usage);
DELETE 1
```

This takes a group query, and produces the rows that didn't make it into the group. And deletes those rows. ctid is likely much more obscure then you need to know for postgres, but it's good information to have.

Finally I showed one more timeseries query within this solar table.

```
unenergy2019=# SELECT country, year, usage FROM solar WHERE country = 'Japan' ORDER by year;
 country | year | usage
---------+------+-------
 Japan   | 1990 |    67
 Japan   | 1991 |    74
 Japan   | 1992 |    81
 Japan   | 1993 |    89
 Japan   | 1994 |    96
 Japan   | 1995 |   106
 Japan   | 1996 |   122
 Japan   | 1997 |   149
 Japan   | 1998 |   189
 Japan   | 1999 |   254
 Japan   | 2000 |   357
 Japan   | 2001 |   547
 Japan   | 2002 |   644
 Japan   | 2003 |   858
 Japan   | 2004 |  1118
 Japan   | 2005 |  1420
 Japan   | 2006 |  1721
 Japan   | 2007 |  1972
 Japan   | 2008 |  2206
 Japan   | 2009 |  2657
 Japan   | 2010 |  3543
 Japan   | 2011 |  4839
 Japan   | 2012 |  6613
 Japan   | 2013 | 12880
 Japan   | 2014 | 22952
 Japan   | 2015 | 34802
 Japan   | 2016 | 50952
(27 rows)
```

Then we went back to Mondial and the extra credit questions to understand queries, joins and the step-by-step process of making a query.

So we went ahead and answered: 5 highest mountains in Greece. First we just need to get a table that has the main information we need: the name of the mountain, the country of the mountain, and the elevation of the mountain. Those are in two different tables, so we need to join them.

```
 unenergy2019=# \c mondial2
You are now connected to database "mondial2" as user "Jon".
mondial2=# \d geo_mountain
                  Table "public.geo_mountain"
  Column  |         Type          | Collation | Nullable | Default
----------+-----------------------+-----------+----------+---------
 mountain | character varying(50) |           | not null |
 country  | character varying(4)  |           | not null |
 province | character varying(50) |           | not null |
Indexes:
    "gmountainkey" PRIMARY KEY, btree (province, country, mountain)

mondial2=# \d mountain
                    Table "public.mountain"
   Column    |         Type          | Collation | Nullable | Default
-------------+-----------------------+-----------+----------+---------
 name        | character varying(50) |           | not null |
 mountains   | character varying(50) |           |          |
 elevation   | numeric               |           |          |
 type        | character varying(10) |           |          |
 coordinates | geocoord              |           |          |
Indexes:
    "mountainkey" PRIMARY KEY, btree (name)
Check constraints:
    "mountaincoord" CHECK ((coordinates).latitude >= '-90'::integer::numeric AND (coordinates).latitude <= 90::numeric AND (coordinates).longitude > '-180

mondial2=# SELECT geo_mountain.country, mountain.name, mountain.elevation
mondial2-# FROM mountain JOIN geo_mountain ON mountain.name = geo_mountain.mountain;
 country |              name               | elevation
---------+---------------------------------+-----------
 GROX    | GunnbjÃ¸rn Fjeld                |      3694
 SVAX    | Newtontoppen                    |      1713
 IS      | Hvannadalshnukur                |      2110
 IS      | Snaefell                        |      1833
 IS      | Hekla                           |      1491
 IS      | Katla                           |      1450
 SF      | Haltiatunturi                   |      1365
 S       | Kebnekaise                      |      2099
```

```
 S       | Sarektjokko                  |      2089
 N       | Higravstinden                |      1146
 N       | Galdhoeppig                  |      2469
 N       | Glittertind                  |      2465
 N       | Snoehetta                    |      2286
 GB      | Ben Nevis                    |      1344
 GB      | Snowdon                      |      1015
 GB      | SgÃ¹rr Alasdair               |       993
 IRL     | Carrauntoohil                |      1041
 GB      | Slieve Donard                |       849
 D       | Feldberg                     |      1493
 D       | Brocken                      |      1141
 D       | Grosser Arber                |      1456
 A       | Zugspitze                    |      2963
 D       | Zugspitze                    |      2963
 F       | Barre des Ecrins             |      4101
 F       | Mont Ventoux                 |      1912
 I       | Gran Paradiso                |      4061
 I       | Gran Paradiso                |      4061
 F       | Mont Blanc                   |      4808
 I       | Mont Blanc                   |      4808
 CH      | Grand Combin                 |      4314
 I       | Matterhorn                   |      4478
```

That join gives us a table with all of the information we need to answer the question. All we need to do now is start filtering. The first thing we want to do is get only the mountains in Greece.

```
mondial2=# SELECT geo_mountain.country, mountain.name, mountain.elevation                                    FROM mountain JOIN geo_mountain
mondial2-# WHERE geo_mountain.country = 'GR';
 country |      name      | elevation
---------+----------------+-----------
 GR      | Smolikas       |      2637
 GR      | Olymp          |      2917
 GR      | Olymp          |      2917
 GR      | Athos          |      2033
 GR      | Kyllini        |      2376
 GR      | Profitis Ilias |      2497
 GR      | Aenos          |      1628
 GR      | Elati          |      1158
 GR      | Dirfi          |      1743
 GR      | Fengari        |      1611
 GR      | Kerkis         |      1433
 GR      | Pilineo        |      1297
 GR      | Pramnos        |      1037
 GR      | Attavyros      |      1215
 GR      | Psiloritis     |      2456
(15 rows)
```

Yay! Suddenly we are so close just by adding that little WHERE statement. Next we just need to order by elevation, and we will see the highest mountains first.

```
 mondial2=# SELECT geo_mountain.country, mountain.name, mountain.elevation
 FROM mountain JOIN geo_mountain ON mountain.name = geo_mountain.mountain
 WHERE geo_mountain.country = 'GR'
 ORDER BY mountain.elevation DESC;

 country |      name      | elevation
---------+----------------+-----------
 GR      | Olymp          |      2917
 GR      | Olymp          |      2917
 GR      | Smolikas       |      2637
 GR      | Profitis Ilias |      2497
 GR      | Psiloritis     |      2456
 GR      | Kyllini        |      2376
 GR      | Athos          |      2033
 GR      | Dirfi          |      1743
 GR      | Aenos          |      1628
 GR      | Fengari        |      1611
 GR      | Kerkis         |      1433
 GR      | Pilineo        |      1297
 GR      | Attavyros      |      1215
 GR      | Elati          |      1158
 GR      | Pramnos        |      1037
(15 rows)
```

So close! A lazy programmer, like myself, would just not worry about the repeat of Mount Olympus (it is repeating because it is located in two provinces). And I would do this.

```
SELECT geo_mountain.country, mountain.name, mountain.elevation
FROM mountain JOIN geo_mountain ON mountain.name = geo_mountain.mountain
WHERE geo_mountain.country = 'GR'
ORDER BY mountain.elevation DESC LIMIT 6;
 country |      name      | elevation
---------+----------------+-----------
 GR      | Olymp          |      2917
 GR      | Olymp          |      2917
 GR      | Smolikas       |      2637
 GR      | Profitis Ilias |      2497
 GR      | Psiloritis     |      2456
 GR      | Kyllini        |      2376
(6 rows)
```

This does technically get me the five tallest mountains in Greece. But since we want to be exacting so our employer feels happy and certain about our programming skills, we simply add DISTINCT to the SELECT command, and that gives us only single instances of all of these columns, thus filtering out duplicates.

```
SELECT DISTINCT geo_mountain.country, mountain.name, mountain.elevation
FROM mountain JOIN geo_mountain ON mountain.name = geo_mountain.mountain
WHERE geo_mountain.country = 'GR'
ORDER BY mountain.elevation DESC LIMIT 5;
 country |      name      | elevation
---------+----------------+-----------
 GR      | Olymp          |      2917
 GR      | Smolikas       |      2637
 GR      | Profitis Ilias |      2497
 GR      | Psiloritis     |      2456
 GR      | Kyllini        |      2376
(5 rows)
```

All done! As I said in class, I highly recommend--especially in the beginnings, but even as you go further in programming--using this step-by-step method. This way you are logging and viewing each stage of the results as you go. This insures that there aren't any hidden mistakes but that you might have missed if you tried to type the whole query at once.