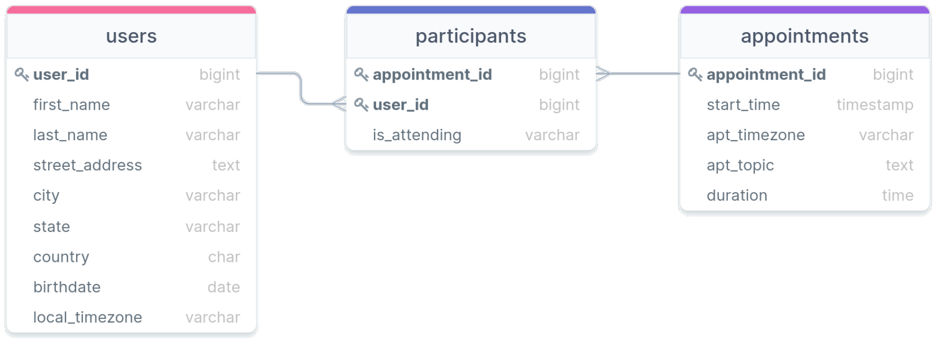
This week you will be taking a look at a set of tables constructed to hold meeting or appointment data for something like a calendar application. You have three tables:

- `users` – Contains information about users of the application, including such information as their name, location, and birthday.
- `appointments` – Contains information about the individual meetings or appointments, including information such as the meeting time (and time zone), duration and topic.
- `participants` – Contains information about which users have been invited to which meetings, and to which they are actually planning on attending.



I have prepared the data for you in two possible methods this week, depending on whichever works best for you. The file `auto_gen.sql` is a database schema dump, so if you want to run it using `psql` like previously, it will take care of creating a new schema (hw8) in your indicated database, and then creating, populating, and adding constraints to the tables. Alternatively, I have included the CSV files for each of the tables and a preparation script named `manual_gen.sql`. It includes the basic table creation commands and constraints, but you would need to import in the data yourself from the provided CSV files.

Follow the link here to accept the assignment and get access to the repository:

Assignment link: https://classroom.github.com/a/y-AmLYBS

1. (4 points) How many users are attending meetings on their birthday? *Answer without using joins for a bonus 2 points!*

2. Unfortunately, some users have committed to appointments that they can't possibly attend, owing to also having committed to another appointment at the same time. We'll start simply here and then build up in complexity. At any time though, it is useful to keep in mind that using `AT TIME ZONE` on a timestamp *without* a current time zone will attach the time zone to that timestamp (essentially creating a `TIMESTAMPTZ` object). Using `AT TIME ZONE` on a `TIMESTAMPTZ` object will return a time **without** a timezone attached but converted to the desired time zone. Both will be useful here. You may also want to look up the Postgres `OVERLAP` keyword, as it can simplify the logical checks here (but it certainly is not necessary).

   (a) (4 points) Consider the case of Simon Smith (`user_id = 12`). What appointments did Simon sign up to *attend* that conflict with another appointment (that he also signed up to *attend*)? Report both the appointment ids and the starting and ending times of each appointment as measured in Simon's local time zone.

   (b) (6 points) Now let's expand things. What are the total number of conflicts for which an individual has signed up to attend two appointments that are overlapping in time? Be careful not to double count here: for a given individual, Event A conflicting with Event B is the same as Event B conflicting with Event A. Multiple individuals could have the same pair of appointments conflicts however, so each of those should be counted. (User 1 having Event A and Event B conflict and User 2 also having Event A and Event B conflict should count as 2.)

3. (4 points) If you looked closely at the appointment topics (`appointments.apt_topic`), you likely realized that they are all generated using only a handful of starting prompts. In particular, the possible starting prompts are:

   - "I am angry about . . . "
   - "Important topic: . . . "
   - "Thoughts on . . . "
   - "I love . . . and you should too!"
   - "Ruminations on the existence of . . . "

   Here, you want to determine how many appointments using each general starting prompt exist in the dataset and output that information in a single table. Let's also order said table by descending number of appointments. So your output might look something like

prompt	count
angry	107
ruminations	⋮
important_topic	⋮
love	⋮
thoughts	⋮

   where I have provided the number of angry appointments to you as a self-check.

4. (10 points) Taking the same theme from the last question, suppose you also wanted to know the distribution of these appointment "categories" over the 7 days of the week. To do so, create a cross-tabulation or pivot table where the appointment categories (angry, ruminations, etc) are down the first column in alphabetical order and the days of the week are across the first row as column headings. Counts of how many appointments of each category are *starting* on each day should be the data represented within the pivot table. You can assume that these start times are determined by the appointment local time zone in each instance. Remember that to use the `crosstab` function, you will first need to ensure that you have installed the `tablefunc` extension to the database where your data is residing. So to be clear, your table should look something like:

topic_type	sun	mon	tues	wed	thur	fri	sat
angry	19	12	14	13	16	15	18
important_topic	⋮	⋮	⋮	⋮	⋮	⋮	⋮
love	⋮	⋮	⋮	⋮	⋮	⋮	⋮
ruminations	⋮	⋮	⋮	⋮	⋮	⋮	⋮
thoughts	⋮	⋮	⋮	⋮	⋮	⋮	⋮

where I have again given you the first row to serve as a sanity check. Export this table to a CSV file named `cat_counts_over_week.csv` and be sure to upload it back to the GitHub repository alongside this problem's template.