ECS 116 Databases for Non-Majors / Data Management for Data Science

Programming Assignment 2
Parts 1 and 2

A. Prelude

1. NOTE: this document includes the material from Part 1, and also new material about Part 2. There will be an additional, smaller Part3 that is focused on (a) visualizing some of your results and (b) providing a few paragraphs that describe why you think you got the results that you did.
2. The assignment is of 25 points.
3. Last date of submission for Parts 1, 2 and 3 of this assignment has been extended to Sunday, May 19, 2024, Friday @ 11:59 pm. You do not have to submit anything for Parts 1 and 2, but we recommend that you try to finish Part 1 by Tuesday, May 7, and Part 2 by Thursday, May 16, so that you have enough time to work on Part 3.
4. This assignment will be in teams of 3 (with one or two teams of 2 or 4 depending on class size).
5. Each team member should have a full installation of the data and the code in their laptop, so that they can run everything. (Teammates might develop different parts of the code, but in the end each teammate should be able to run everything and get similar results – although behavior on Macs vs. PCs may differ). In particular, each teammate will be required to create the three json files described in Step 3 below, and also the visualizations described in Step 4, all based on execution on their own machine. (Each team can submit a single document that includes (a) the visualizations generated by each team member and (b) the jointly written paragraphs describing the results obtained.)
6. Late submissions will be graded according to the late policy. Specifically, 10% of grade is deducted if you are up to 24 hours late, 20% is deducted if you are 24 to 48 hours late, and no credit if turned in after 48 hours.
7. Plagiarism is strictly prohibited. You're free to discuss high-level concepts amongst your peers. However, cheating will result in no points on the assignment and reporting to OSSJA.

B. Step 1: Uploading the AirBnB data for New York City into PostgreSQL

1. Download the 4 csv files in the GoogleDocs at
https://drive.google.com/drive/folders/14gWh0ck3vzWxyakaWHHH38AgWY7UC-IQ?usp=sharing
2. In DBeaver create a new database **airbnb**.
3. Upload the csv files for calendar, listing and neighborhoods into PostgreSQL.
    - As with the africa_ac_db.csv file, after a first attempt to upload one of these csv files into DBeaver you may have to modify the data types of various columns, then delete the data, and reload the data directly into the table in DBeaver. In general, for an "id", "listing_id" or similar fields use a data type "varchar(32)" (or perhaps larger if necessary). For any date fields use data type "date", so that you can make queries based on date ranges.
4. As a sanity check, you can confirm that the number of records in your database tables correspond to the number of records in the csv files:
    - Calendar should have 14,299,870 records
    - Listing should have 39,202 records
    - Neighborhood should have 230 records

- NOTE: you can check the number of lines in a csv file from a terminal window as follows:
  - For mac, in a terminal window type, e.g., `wc <filename>`. This gives the line count, the word count and the character count
  - For PC, in a powershell window type `type <filename> | find /v /c ""`. (See https://jordenthecoder.medium.co to-count-number-of-lines-of-large-data-files-in-windows-660ae8a3f4f7.)

5. Now work on uploading the file reviews.csv into PostgreSQL

   - On the first attempt to upload using DBeaver you will get errors talking about "integer out of range". Use the "Skip All" choice so that DBeaver will create the relation schema for reviews
   - Check on the table - how many tuples does it have in it? What are the columns and data types?
   - Since all of the integer columns are some form of ID, let's convert all of them to varchar(32)
   - Empty the table and reload the csv file
   - Now you get an issue about character length of the "comments" attribute. Resolve this by
   - Now DBeaver complains about a row that appears corrupted, in particular because it looks like part of a comment is in the listing ID. Note also that if you do "skip" then the entire batch holding the offending tuple will be dropped.
   - At this point it probably makes sense to try a different approach for loading the csv file!
   - Try using the sql command COPY that loads directly from a csv file.

     ```
     COPY reviews(listing_id, id, date, reviewer_id, reviewer_name, comments)
     FROM '<full path to csv file>'
     DELIMITER ','
     CSV HEADER;
     ```

   - Notice the error that a comment is > 4096 characters. Make that datatype `varchar(10000)` and try again.
   - Finally, examine the data type and values in the "date" columns of the calendar table and the reviews table. Change the data type in reviews to match the data type in calendar. This will enable selections joins on date fields.
   - NOTE: In the above steps you used both DBeaver and the COPY command to upload data. Sometimes one works, sometimes the other, so we wanted you to have practice with both.

C. Step 2: Setting up test harness and related infrastructure

1. The goal of this step is to set up a "test harness", that is, an environment that will make it easy to run a range of difference performance tests (also called "benchmarking"). The testing will involve

   - A handful of queries and updates against the AirBnB data, and
   - A handful of indexes that can be added to (and dropped from) the AirBnB data.

   During the benchmarking you will be exploring the impact (positive or negative) of having some indexes in place, when executing both queries and updates.

2. To set up your test harness, you should follow the examples in the notebook Prog-ass-2–v04–mostly-without-util.ipynb, which is in the PROG-ASSMT-2 area of Canvas. The notebook is self-explanatory. Also, Aunsh has gone over the notebook in the Discussion of May 2, 2024.

D. Step 3: Doing performance testing & capturing performance results

1. For this part, you are to write code that will create three separate json files, to be called

   - listings_join_reviews.json
   - text_search_query.json

- update_datetimes_query.json

Output from runs on Rick's mac and Aunsh's pc will be posted into Canvas. (The file names used by Rick and Aunsh will not strictly follow the above naming conventions.)

2. In the Prog-ass-2–v04–mostly-without-util.ipynb notebook mentioned in Step 2 above, the performance profile information for each group of runs for a query included

   - avg
   - min
   - max
   - std

   Going forward, please add two additional kinds of information, specifically

   - count (which holds the number of runs you did for the particular query and particular index configuration)
   - timestamp (which should have the format `'2024-05-07-23:37:50'` (which should be the date/time that this particular run was performed on your machine.)

3. For all three of the performance tests, you are to execute each query + index configuration 50 times. This is to try to get the standard deviation of the execution times to be a reasonably low number. For some of the update queries the running time for executing the query in the 4 index configurations may take up to an hour, so please plan ahead. Also, do follow the pattern shown in the notebook Prog-ass-2–v04–mostly-without-util.ipynb that enables your code to store the results from the 50 executions of a single query with its index configurations. This will enable you to break up the creation of your json files (especially the third one).

4. When you are running the tests you may continue to use your laptop, but you should avoid doing other compute-intensive jobs. Such jobs will slow down the query processing in unpredictable ways and may skew your results.

5. Suggestion: As you are developing and testing your code, use a small "count", i.e., a small number of executions for each query + index configuration. E.g., use count = 3 or count = 5. Once things are debugged, then do your runs with count = 50.

D..1 Step 3a: Testing with query that joins listings and reviews (with a new column)

1. For this part you will testing queries very similar to the ones called `q_listings_join_reviews_YYYY`. However, you will modify the `reviews` table by adding a new column `datetime` of type `timestamp`. (We are using timestamp-based data because there is a simple way to update it, which is the focus of Step 3c below.)

2. In particular you should do the following steps:

```
alter table reviews
add column datetime timestamp;

update reviews
set datetime = TO_TIMESTAMP((TO_CHAR(date, 'YYYY-MM-DD') || ' 12:00:00'),
                    'YYYY-MM-DD hh24:mi:ss')::timestamp without time zone;
-- this took almost 3 minutes on Rick's mac
```

   This has the effect of adding a new column, and then populating it to be the same as the `date` column, but with a time of day (specifically, 12 noon), and without having a timezone.

3. You should change your queries `q_listings_join_reviews_YYYY` to work with `datetime` rather than `date`. Please note that conditions such as `datetime <= '2015-12-31'` still work. (A nuance is that that particular condition will not be satisfied by the `datetime` value `'2015-12-31 12:00:00.000'`, but we will not worry about that for this exercise.) Also, rather than using an index `date_in_reviews` you should use `datetime_in_reviews`

4. Please run tests for the the queries `q_listings_join_reviews_YYYY` for all 4 combinations of indexes drawn from

- `datetime_in_reviews`
- `id_in_listings`

(Having no indexes is one of the combinations.)

5. Please use the naming conventions and format illustrated in the notebook Prog-ass-2–v04–mostly-without-util.ipynb for your file listings_join_reviews.json.

6. As you look at the results obtained, you may want to consider the number of reviews that happened in each year. You should create a group-by query that lists, for each year, the number of reviews in that year. This will be useful when you try to understand and explain your results in Step 5 below.

D..2 Step 3b: Testing with query that does text search

1. This part will involve searches into the `comment` field of `reviews` for specific words. We will use a technique known as "ts-vectors" (ts is for "text search") which will be explained in the 2024-05-09 lecture. (Also, a somewhat good explanation of ts-vectors may be found at https://medium.com/geekculture/comprehend-tsvector-and-tsquery-in-postgres-for-full-text-search-1fd4323409fc.)

2. To set things up, you will add another column to reviews, populate it, and add a Generalized Inverted Index (GIN) as follows:

```
alter table reviews
add column comments_tsv tsvector

update reviews
set comments_tsv = to_tsvector(comments)

CREATE INDEX IF NOT EXISTS comments_tsv_in_reviews
ON reviews using GIN (comments_tsv);
-- took about 10 seconds on Rick's mac
```

The new column is pretty interesting – check it out!

3. As explained in lecture, to use the ts index to search for occurrences of the word 'awesome' (or more accurately, for the lexeme 'awesom') you can use the following query (which also restricts attention to reviews in 2015).

```
select count(*)
from reviews r
where comments_tsv @@ to_tsquery('awesome')
  and datetime >= '2023-01-01'
  and datetime <= '2023-12-31'
```

4. Note: Once you have created the index `comments_tsv_in_reviews` you will not drop it. In experiments in which you want to compare performance of having or not having the index `comments_tsv_in_reviews`, you will use 2 different queries. The query for testing performance with the index will be like the one shown above. The query for testing performance without the index will go against the `comments` field. In particular, you should use queries of the form:

```
select count(*)
from reviews r
where comments ILIKE '%awesome%')
  and datetime >= '2023-01-01'
  and datetime <= '2023-12-31'
```

(Using "ILIKE" makes the pattern matching case insensetive. When you use "LIKE" the pattern matching is case sensetive.)

- IMPORTANT NOTE: if you pass a query like this one into sqlalchemy for evaluation, you need to use `'%%'` rather than `'%'`. I.e., you should use this expression:

```
select count(*)
from reviews r
where comments ILIKE '%%awesome%%')
  and datetime >= '2023-01-01'
  and datetime <= '2023-12-31'
```

5. Please run experiments that use queries such as the two just shown, with the following variations:

   - Run them for all years in [2009, 2010, 2011, 2012, 2013, 2014, 2017, 2019, 2023]. (The number of reviews for these years is representative of the number across all years from 2009 to 2024.)
   - Run them for the words: `'horrible'`, `'awesome'`, and `'apartment'`.

   (These 3 words were chosen to illustrate the query performance in 3 different situations. What makes these 3 words different? The answer to this question will be relevant to Step 5 below.)

6. Please run experiments for the above queries based on

   - using the index `comments_tsv_in_reviews` vs. not using an index and doing searches in the `comments` field
   - using or not using the index `datetime_in_reviews`

7. In your file text_search_query.json, use names such as `awesome_2015` for the different queries used.

D..3 Step 3c: Testing with updates to the datetime field of reviews

1. This set of experiments will explore whether having an index on `datetime` leads to a performance loss when updating values of `datetime`.

2. There are two challenges when trying to do experiments that involve updating values

   - Finding an easy, repeatable way to make updates. For the `datetime` field this is easy because the `timestamp` data type supports some arithmetic operations. In particular, if you want to add or subtract 5 days to a `timestamp` value, you can use queries such as the following:

```
UPDATE reviews r
SET datetime = datetime + interval '5 days'
FROM listings l
WHERE l.id = r.listing_id
  AND l.neighbourhood_group = 'Manhattan'
RETURNING 'done';

UPDATE reviews r
SET datetime = datetime - interval '5 days'
FROM listings l
WHERE l.id = r.listing_id
  AND l.neighbourhood = 'Bedford-Stuyvesant'
RETURNING 'done';
```

     Note that one of these queries is using a `neighbourhood_group` value (namely, 'Manhattan') and the other is using a `neighbouhood` value (namely, 'Bedford-Stuyvesant'). These were chosen because different numbers of reviews are related to AirBnB units in those two areas.

     (We could not think of a simple way to update the `comments` field of the `reviews` table – if you have a good idea please let us know.)

   - The second challenge is to find a way to systematically undo updates that are made. It is desirable to undo the updates, so that you are basically always starting from the same base data sets.

In the case of the `datetime` field, we can make and unmake updates by alternately adding 5 days and subtracting 5 days from specified `datetime` values. In your code we recommend that as you invoke specific update commands, you alternate between adding 5 days and subtracting 5 days. (Adding or subtracting should give the same performance results.) If you do an even number of invocations, then your data should be back to the original value after all execution is done.

3. For this testing, you should use a family of update queries, similar to the two shown just above. You should make queries based on the following neighbourhoods and neighbourhood_groups:

   - neighbourhoods: New Springville, Fort Hamilton, Long Island City, Bedford-Stuyvesant
   - neighbourhood_groups: Staten Island, Bronx, Quuens, Manhattan

   (What makes these choices interesting to compare?)

4. You should run these tests with all 4 combinations involving the following indexes:

   - `datetime_in_reviews`
   - Also:
     - for neighborhood values: `neighbourhood_in_listings`
     - for neighborhood_group values: `neighbourhood_group_in_listings`

5. Please use names for these updates with the following form: `update_datetimes_query_<name of area>`, e.g., `'update_datetimes_query_Manhattan'` or `'update_datetimes_query_Long Island City'`

E. Step 4: Visualizing performance results

(Will be filled in for Part 3.)

F. Step 5: Explanation of performance results

(Will be filled in for Part 3.)

G. Submission

(Will be filled in for Part 3.)