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

Programming Assignment 3

Prelude

1. The goal of this programming assignment is to provide some hands-on experience with MongoDB, one of the early and still widely used NoSQL databases. You have already done part of the work towards this assignment in Problem Set 5. This Programming Assignment builds on Problem Set 5 in a couple of directions.

2. The assignment is of 25 points (but it may receive less weight than Programming Assignment 2 because it is significantly shorter).

3. The assignment is due Sunday, June 9, 2024, at 11:59 pm.

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 on their machine). In particular, each teammate will be required to create the json files and the csv file described below, all based on execution on their own machine.

6. Each team can create a single document that includes the jointly written paragraphs describing the results obtained (Part 6).

7. Step 5 is "Extra Credit". This part is optional. If you perform the steps there then you can get some extra credit towards your overall course grade.

8. As with Programming Assignment 2, ChatGPT and/or other LLMs can be used, and we ask that you give a brief statement about how it was used and what your experience was.

9. 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.

10. 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.

A. Step 1: Creating MongoDB collection holding listings with embedded reviews, using df and dict operations

This step is identical to Problem Set 5. If you have completed Problem Set 5, then you are done with this part, except please rename your notebook as "1–Building-listings-with-reviews-using-python.ipynb". This will be submitted as part of your zip file for this assignment.

If not, then you should perform Problem Set 5, and name your final notebook "1–Building-listings-with-reviews-using-python.ipynb".

Recall that for Step 1 (i.e., Problem Set 5), you brought slightly pre-processed data from PostgreSQL into your PyMongo environment, and used python and pandas to further format the data for insertion into a MongoDB collection. That approach was not time efficient – it may have taken around 50 to 70 minutes for some of the processing. In the following you will do things primarily within the MongoDB system, and primarily using the aggregate operation with pipelines. You will see that if you can do something within MongoDB, then it is probably much for time efficient.

B. Step 2: Creating MongoDB collection holding listings with embedded calendar availability, using an aggregation pipeline

For this step. please refer to the notebook "2–Loading Local MongoDB with calendar csv data–vXX.ipynb" in the Programming Assignment 3 folder in Canvas. That notebook shows how you can load the calendar.csv file into a dataframe, and from there into mongodb. In the notebook, the resulting collection is called "calendar". The datatypes for fields holding dates should be datetime, and for any boolean fields should be Boolean.

Your goal is to build a notebook, called "2–Building-listings-with-calendar-using-aggregate.ipynb", that creates another collection, called "listings_with_calendar" which includes a document for each listing. The document should have fields for

1. `_id`: holds the listing_id value of the listing

2. `average_price`: holds the average price across all of the records in calendar.csv associated with this listing, having type numeric (integer or float is OK)

3. `first_available_date`: holds the minimum date of any calendar record associated with the listing, having type datetime

4. `last_available_date`: holds the maximum date of any calendar record associated with the listing, having type datetime

5. `dates_list`: Holds a list of dictionaries, one for each calendar entry associated with the listing. Each dictionary should include the following fields:

   (a) `date`, of type datetime

   (b) `available`, of type Boolean

   (c) `price`, of type numeric (not string)

   (d) `minimum_nights`, of type integer

   (e) `maximum_nights`, of type integer

The notebook includes some code illustrating how you can check the data types of documents in a MongoDB collection. It also includes a function `convert_lwc_to_json` that illustrates how you can convert MongoDB documents into python dictionaries that can be written into json files.

For this step of the Programming Assignment, you are to:

1. Create a pipeline specification that will build the listings_with_calendar collection, and use that pipeline as part of the notebook to create the collection. Here are some notes:

   (a) In one approach to building a pipeline that works, the first step is a `$group` operator. That can be used to define how the scalar fields are to be populated. Also, to populate the `dates_list` field you can use the `$push` operator, which forms an array of all elements that are being grouped.

   The second (and final) step is to use the `$out` operator to write the result of the aggregation into the collection `listings_with_calendar`.

   (b) When you are working to define the pipeline, you may want to work with a small collection that corresponds to, e.g., of the first 5000 documents in `calendar`.

   (c) Your collection `listings_with_calendar` should hold 39,201 documents. (Why is that one less that the number of documents in the collection `listings_with_reviews` that you built for Problem Set 5?)

2. Select a subset of the collection which holds documents for all listings whose `id` has prefix '`1000`', convert these documents into something that can be written into a json file, and write them into a file named `listings_with_calendar_subset.json`'. This file is to be included into your zip submission. Here are some notes:

   (a) The notebook "2–Loading Local MongoDB with calendar csv data–vXX.ipynb" provides an illustration of how to convert data from MongoDB into dictionaries that can be written out to json files.

   (b) You can find a json file similar to the file you are to produce in Canvas in the Programming Assignment 3 folder.

   (c) Your file should hold 43 documents.

C. Step 3: Creating MongoDB collection holding listings with embedded reviews, using an aggregation pipeline

In this step you will revisit the goal of Step 1 (i.e., Problem Set 5), which was to build a MongoDB collection `listings_with_reviews`. But for this step, you will use the aggregate function rather than doing things with pandas and python.

Specifically, you are to build a notebook called "3–Building-listings-with-reviews-using-aggregate.ipynb" that:

1. Imports the listings.csv and reviews.csv files into dataframes

2. Puts both of them into MongoDB collections. (Please ensure that date columns are converted to the datetime data type, and address issues with NaT.)

3. Using an aggregation pipeline, builds a collection `listings_with_reviews_m`. This should hold data very similar to the collection `listings_with_reviews` that you built for Step 1. (There are some minor differences because of some operations performed in Step 1 vis-a-vis some operations performed here. Can you find them?)

   Some notes:

   (a) One way to build the pipeline would be to start with a `$lookup`, and then use `$out` to write the output into the target collection

   (b) IMPORTANT NOTE: To make your pipeline run quickly (e.g., in about 7 to 15 seconds), you should create an index on 'listing_id' in your db.reviews collection. You can use a command such as the following:

   ```
   db.reviews.create_index('listing_id')
   ```

   (If you don't have this index, your pipeline would probably run for 2 to 4 hours!)

4. Finally, produce a file "listings_with_reviews_m_subset_1000.json" that holds documents that correspond to the documents in your `listings_with_reviews_m` collection whose `id` value has prefix "1000". Some notes:

   (a) In addition to dealing with the `datetime` values, you will have to modify the `ObjectID` values (make them strings) and the `NaN` values (test for that using `math.isnan` operator, and map to `None`).

   (b) You can find a json file similar to the file you are to produce in Canvas in the Programming Assignment 3 folder.

   (c) Your file should hold 43 documents, as in Step 2.

D. Step 4: Creating MongoDB collection holding listings with embedded data for both reviews and calendar availability

For this step, you are to create a notebook named "4–Building-listings-with-reviews-and-cal.ipynb" that forms a kind of join of your collections `listings_with_reviews_m` and `listings_with_calendar`, and puts it into a collection called `listings_with_reviews_and_cal`. In particular, each document in the collection `listings_with_reviews_and_cal` should include information about a listing, including

1. all scalar fields about that listing from both `listings_with_reviews_m` and `listings_with_calendar`, except for the `_id` field from `listings_with_calendar`.

2. a field `reviews`, holding the array of data about reviews associated with the listing

3. a field `dates_list` holding the array of data about calendar entries associated with the listing.

Here are some notes about one way to build a pipeline to create the collection `listings_with_reviews_and_cal`.

1. Run the aggregate command on the `listings_with_reviews_m` collection.

   Remember that `listings_with_reviews_m` has 1 listing that `listings_with_calendar` does not have. BTW, the id of that listing is '35384734'.

2. Start the pipeline with a `$lookup` that will form, intuitively speaking, something close to the left join of `listings_with_reviews_m` and `listings_with_calendar`. In my `$lookup`, I used the name `cal_docs` for holding the array of docs from `listings_with_calendar`.

3. Now use the `$unwind` operator on the `$cal_docs` field. This has the effect of breaking each `$cal_docs` array into separate documents. The intermediate result after the `$unwind` is quite close to being the left join of `listings_with_reviews_m` and `listings_with_calendar`. In order to retain data about the listing with id '35384734', you need to use the following formulation

```
{ '$unwind': { 'path': '$cal_docs',
               'preserveNullAndEmptyArrays' : True
             }
},
```

4. Now use an `$addFields` operator to add in the fields for `average_price`, `first_available_date`, `last_available_date`, and `dates_list`. For each of these you will need a formulation something like:

```
'first_available_date': '$$ROOT.cal_docs.first_available_date',
```

   What is `$$ROOT` here? This step of the pipeline is basically operating on a stream of documents. For a given document, `$$ROOT` refers to the root of that document.

5. Now use an `$unset` operator to remove the `cal_docs` field.

6. Finally, use `$out` to write the output of the pipeline into the collection `listings_with_reviews_and_cal`.

As with steps 2 and 3, you are to produce a json file called "listings_with_reviews_and_cal_subset_1000.json" that holds documents that correspond to the documents in your `listings_with_reviews_and_cal` collection whose `id` value has prefix "1000".

The file "listings_with_reviews_and_cal_subset_111.json" holds analogous data for listings whose id starts with '111'.

E. EXTRA CREDIT: Step 5: Queries for "good" and "bad" listings, with and without text index

In this step you will learn a little about the text indexing capabilities in MongoDB Community Edition. (The text indexing for the cloud-hosted MongoDB Atlas is a bit stronger). In particular, we focus on finding listings that appear to have at least one positive review, based on the presence of one or more of a subset of very positive words, and also listings that appear to have one negative review. (Of course, this is only scratching the very tip of the iceberg in terms of reviews analysis.)

The list of "superlative words" that you should use is

```
superlative_words = ['astounding',
                     'amazing',
                     'awesome',
                     'excellent',
                     'exceptional',
                     'extraordinary',
                     'fantastic',
                     'great',
                     'magnificent',
                     'splendid',
                     'wonderful']
```

Also, the list of "super negative words" that you should use is

```
super_negative_words = ['aweful',
                        'horrible',
                        'terrible']
```

(A main point is that there will be many hits for the superlative words, and not too many hits for the super negaive words.)

For this step you are to create a notebook called "5–Finding-good-listings.ipynb", that works with the collection `listings_with_reviews_and_cal`. You will also keep track of some counts and some runtimes, and submit a csv with this information (see below).

The notebook should include the following things:

1. Create of a query, called "Query 5 pos" that uses `$regex` conditions to find all listing documents for which there is at least one review that contains at least one of the superlative words. Keep track of how many listings satisfy this condition, and how long it takes to run. Some notes:

   (a) When you build Query 5 pos, you may want to take advantage of the fact that query expressions in pymongo are actually dictionary objects. So you can use python to create a dictionary object that incorporates all of the superlative words. This would be a good software engineering practice, because in the real world, the set of superlative words would probably expand over time.

   (b) To keep track of the time it takes to answer the query you might use a formulation like this (where "condition" holds a python dictionary that expresses the condition about superlative words occurring in reviews):

   ```
   time1 = datetime.now()
   result = db.listings_with_reviews_and_cal.find(condition)
   time2 = datetime.now()
   print(f'The time taken for the selection was {util.time_diff(time1,time2)} seconds.')

   time3 = datetime.now()
   ```

```
l = list(result)
time4 = datetime.now()
print(f'\nThe time taken to create the list was {util.time_diff(time3,time4)} seconds.')
```

On Rick's machine the results were

```
The time taken for the selection was 0.000165 seconds.

The time taken to create the list was 41.951336 seconds.
```

Why do you think that creating the `result` (a cursor) is so quick but creating `list(result)` takes so only?

(c) The number of listings you get should be between 25000 and 25500, and end with a 6.

2. Create a query, called "Query 5 neg" which is analogous to Query 5 pos, but using the super negative words. Some notes:

   (a) You should get a number of hits between 1500 and 2000, and ending in 2.

3. Create a query, called "Query 6 pos" that extends Query 5 pos by adding two more conditions, namely:

   - The listing is available on or after February 1, 2025, and
   - The average price of the listing is $200 or less

   Run this query, and keep track of the time it takes and the number of hits.

   Some notes:

   (a) The number of hits should be between 19000 and 19500, and end with 8

4. Create a query, called "Query 6 neg", which is analogous to "Query 6 pos" but using the negative words. Keep track of the number of hits and the time it takes. Some notes:

   (a) The number of hits is between 1100 and 1500, and ends with 1.

5. Set up a text index on the comments in all of the reviews arrays. To do this you can use a formulation such as

   ```
   index_name = db.listings_with_reviews_and_cal.create_index( { 'reviews.comments' : 'text' } )
   ```

   If you run this command, then index_name will be assigned the value "`reviews.comments_text`" If you want to drop this index, you could use the formulation

   ```
   db.listings_with_reviews_and_cal.drop_index('reviews.comments_text')
   ```

   To drop all indexes from this collection, you can use

   ```
   db.listings_with_reviews_and_cal.drop_indexes()
   ```

   Also, you can get information about the indexes on a collection using the following.

   ```
   cursor = db.listings_with_reviews_and_cal.index_information()
   cursor1 = db.listings_with_reviews_and_cal.list_indexes()

   for i in cursor:
       print(i)
   ```

```
print()
for i in cursor1:
    print(i)
```

Unlike the text indexing in PostgreSQL, the text indexing in MongoDB focuses on words rather than lexemes. The text indexing in MongoDB is case insensitive, and also diacritic insensitive (i.e., it ignores umlauts and other markings on letters).

Keep track of the time it takes to create this index.

6. Create a query called "Query 7 pos", which takes advantage of the new index, and similar to Query 5 pos, finds all listings that have at least one review that includes at least one of the superlative words. To use the text index for such a query, you can use a formulation such as the following (but including all of the superlative words)

```
condition_ind = { '$text' : { '$search': 'awesome amazing' } }

result = db.listings_with_reviews_and_cal.find(condition_ind)
```

Note: MongoDB Community Edition allows for only one text index, and so the use of `$text` in the above formulation is not ambiguous.

Keep track of how long it takes to run the query, and how may hits you have. Some notes:

   (a) The number of hits is between 25000 and 30000, and ends with a 7. The number is slightly different than the number of hits using the `$regex` query. Why do you think this is?

7. Finally, create a query called "Query 7 neg", analogous to Query 5 neg, but using the text index. Keep track of how many hits you have, and the time it takes. A note:

   (a) The number of hits should be between 1500 and 2000, and end in 0.

8. Prepare a csv file that reports on the results you obtained in the previous steps. In particular, the csv file should hold a table that looks like the following. (The number of seconds should have at least 3 or 4 decimal digits, and you do not need to round them.)

Action	**Count**	**Time in Seconds**
Query 5 pos	< count >	< number of secs >
Query 5 neg	< count >	< number of secs >
Query 6 pos	< count >	< number of secs >
Query 6 neg	< count >	< number of secs >
Create text index		< number of secs >
Query 7 pos	< count >	< number of secs >
Query 7 neg	< count >	< number of secs >

Table 1: Example structure of table in the csv file to be submitted

(You can create the csv file by hand, i.e., you do not need to have your notebook produce it.)

Note: If we were serious, then we would be running all of these actions about 50 times and using the average running times. But the quarter is ending ...

F. Step 6: Comments about what you observed

These will be provided by Tuesday evening, June 4.

G. Submission Instructions

These will be provided by Tuesday evening, June 4.