Report – Assignment 4 on Document Databases

This report explains the process of creating a document database, forming relationships between database collections, and subsequently querying it. It also includes exploring and visualizing query plans and examines creating indexes to improve query performance.

Note: The program code is provided individually for each question in the root folder. The program can be executed by following the README.md instructions. The questions from the root directory contain method calls from the other files in the “funcs” subfolder to make the code more manageable and readable. The global settings are provided in “globals.py” inside the “funcs” subfolder.

Question 1: Provide a program to load the data from your relational database or the original files into MongoDB.

The program in “q1.py” loads the data from the original files into MongoDB. The final document structure matches the one provided in the assignment questions file. The insertion follows the following sequence.

1. Insert users by reading users XML file.
2. Insert badges inside users by reading the badges XML file. (Mapping is used to insert with valid user object ID)
3. Insert posts and tags by reading the posts XML file.
4. Insert comments inside posts using the comments XML file. (Mapping is used to insert with valid post object ID)

From the previous explorations and the program provided in “analyze_data.py”, I ensure that only cleaned records (non-nulls) are inserted. Based on the analysis of input files, I skipped the data cleaning where all values were present (users, badges) based on the manual analysis. In cases, where some attributes contained missing values (e.g., in posts), I cleaned data by removing those records. After that, posts are filtered (Owner User ID) based on valid users to keep only the valid posts. Likewise, comments were also filtered (User ID) based on valid users. I mapped the ID attributes of posts and comments with users by using MongoDB object IDs for creating proper references. After the mapping, I removed the keys used for mapping (ID, User ID, and Post ID). This ensures that the final database only contains the relevant and non-null data as the structure shown below (as required in assignment task). The insertion takes around 1445 seconds (around 24 minutes) across multiple runs on my machine.

```
Posts:
{       _id : _,
        OwnerUserId: _,
        PostTypeId : _,
        CreationDate : _,
        Title : _,
        Body : _,
        ViewCount : _,
        Score : _,
        Tags : [iptables, ekiga, …],
        Comments : [ {
                UserId : _, Text: _,
                CreationDate: _, Score: _
        }, … ]
}
```

```
Users:
{       _id : _,
        DisplayName : _,
        CreationDate : _,
        LastAccessDate : _,
        Views: _,
        Badges: [ {
                Name: Student,
                Date: 2010-07-28},
        … ]
}
```

Figure 1. Final schema for collections and attributes for the assignment tasks.

The inserted statistics are as under;

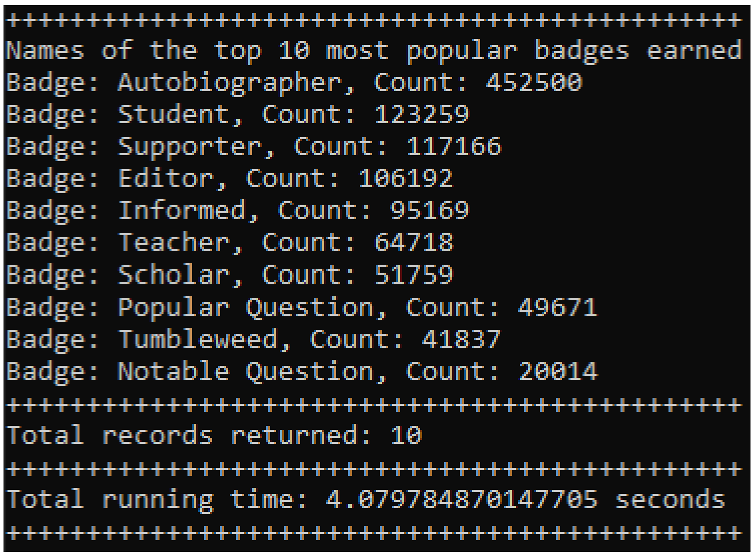
Users Inserted: 1450497, Posts Inserted: 404814, Badges Inserted (inside users): 1973752, Tags Inserted (in posts): 1135530, Comments Inserted (inside posts): 806941

Question 2: Provide a program to run queries over the MongoDB database. Report evidence that you retrieved what was expected along with the time each query took to run.

The queries are provided in the "queries.py" in the "funcs" subfolder. Queries are named as methods with query number (i.e., the answer to the first query in this question is named "q2_query1()", and so on). The program can be run from the root folder by running the "q2.py" file. The following provides details on each query's result and execution time.

Query 1. Names of the top 10 most popular badges earned by users within a year of creating their accounts.

Time Taken: 4.07 seconds, Records: 10

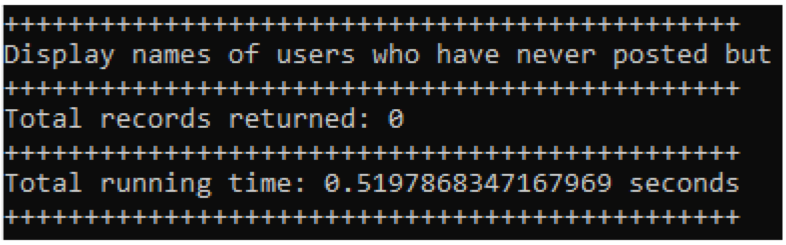


```
++++++++++++++++++++++++++++++++++++++++++++++
Names of the top 10 most popular badges earned
Badge: Autobiographer, Count: 452500
Badge: Student, Count: 123259
Badge: Supporter, Count: 117166
Badge: Editor, Count: 106192
Badge: Informed, Count: 95169
Badge: Teacher, Count: 64718
Badge: Scholar, Count: 51759
Badge: Popular Question, Count: 49671
Badge: Tumbleweed, Count: 41837
Badge: Notable Question, Count: 20014
++++++++++++++++++++++++++++++++++++++++++++++
Total records returned: 10
++++++++++++++++++++++++++++++++++++++++++++++
Total running time: 4.079784870147705 seconds
++++++++++++++++++++++++++++++++++++++++++++++
```

Query 2. Display names of users who have never posted but have a reputation greater than 1,000.

As the schema does not contain the "reputation" attribute as shown in question 1, therefore, it is expected that query will return no results.

Time Taken: 0.51 seconds, Records: 0

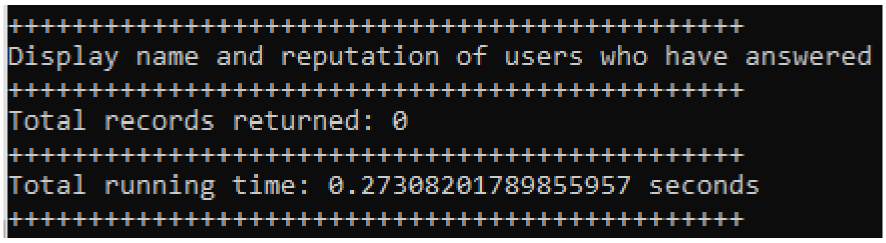


```
++++++++++++++++++++++++++++++++++++++++++++++
Display names of users who have never posted but
++++++++++++++++++++++++++++++++++++++++++++++
Total records returned: 0
++++++++++++++++++++++++++++++++++++++++++++++
Total running time: 0.5197868347167969 seconds
++++++++++++++++++++++++++++++++++++++++++++++
```

Query 3. Display the name and reputation of users who have answered more than one question with the tag "postgresql".

As the schema does not contain the "reputation" attribute as shown in question 1, the, projection skips it.

Time Taken: 0.27 seconds, Records: 0

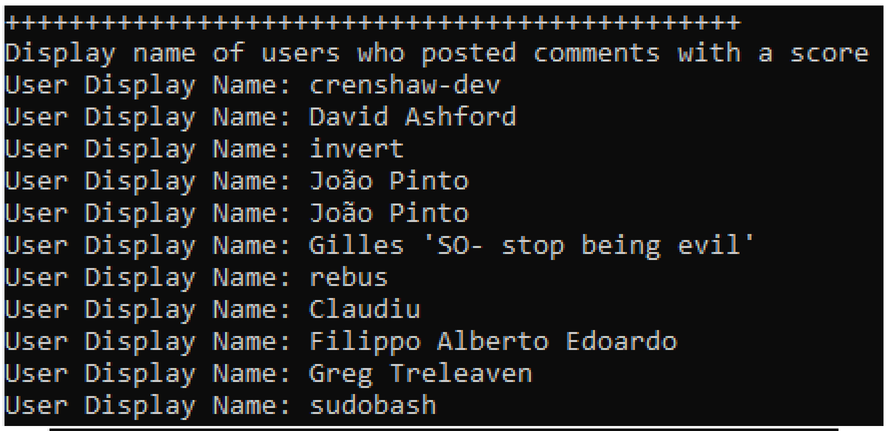


```
++++++++++++++++++++++++++++++++++++++++++++++
Display name and reputation of users who have answered
++++++++++++++++++++++++++++++++++++++++++++++
Total records returned: 0
++++++++++++++++++++++++++++++++++++++++++++++
Total running time: 0.27308201789855957 seconds
++++++++++++++++++++++++++++++++++++++++++++++
```

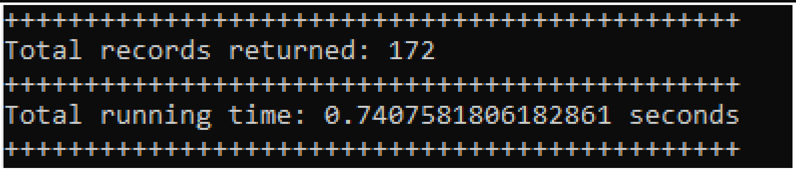
Query 4. Display the names of users who posted comments with a score greater than 10 within the first week of creating their accounts.

The query does not require to sort the results, therefore the sequence of results may vary on subsequent runs.

Time Taken: 0.74 seconds, Records: 172



```
++++++++++++++++++++++++++++++++++++++++++++++
Display name of users who posted comments with a score
User Display Name: crenshaw-dev
User Display Name: David Ashford
User Display Name: invert
User Display Name: João Pinto
User Display Name: João Pinto
User Display Name: Gilles 'SO- stop being evil'
User Display Name: rebus
User Display Name: Claudiu
User Display Name: Filippo Alberto Edoardo
User Display Name: Greg Treleaven
User Display Name: sudobash
```

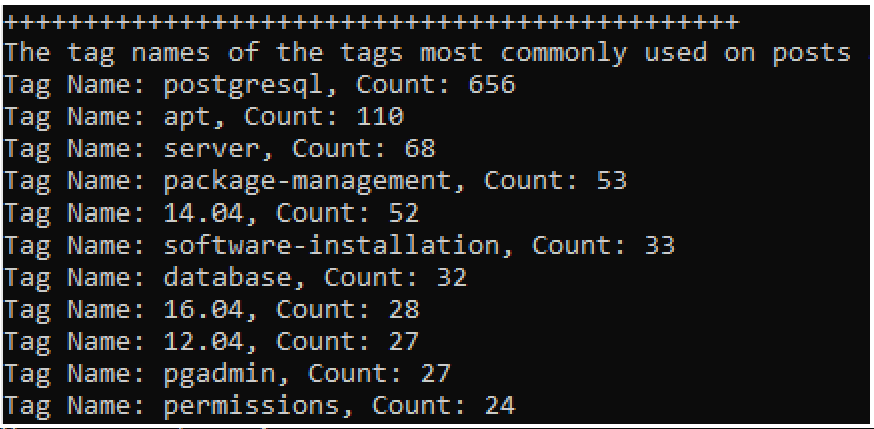


```
++++++++++++++++++++++++++++++++++++++++++++++
Total records returned: 172
++++++++++++++++++++++++++++++++++++++++++++++
Total running time: 0.7407581806182861 seconds
++++++++++++++++++++++++++++++++++++++++++++++
```

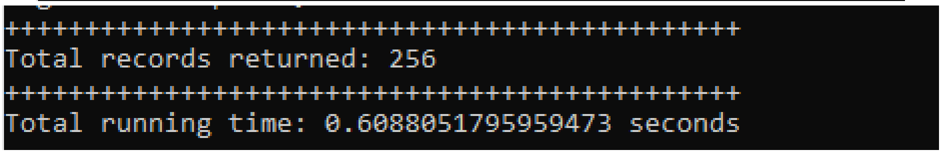
PTO

Query 5. The tag names of the tags most commonly used on posts along with the tag "postgresql" and the count of each tag.

Time Taken: 0.60 seconds, Records: 256



```
++++++++++++++++++++++++++++++++++++++++++++++
The tag names of the tags most commonly used on posts
Tag Name: postgresql, Count: 656
Tag Name: apt, Count: 110
Tag Name: server, Count: 68
Tag Name: package-management, Count: 53
Tag Name: 14.04, Count: 52
Tag Name: software-installation, Count: 33
Tag Name: database, Count: 32
Tag Name: 16.04, Count: 28
Tag Name: 12.04, Count: 27
Tag Name: pgadmin, Count: 27
Tag Name: permissions, Count: 24
```



```
++++++++++++++++++++++++++++++++++++++++++++++
Total records returned: 256
++++++++++++++++++++++++++++++++++++++++++++++
Total running time: 0.6088051795959473 seconds
```

Question 3: Briefly explain the execution plan for each of the previous queries.

The explained queries are provided in the "queries.py" in the "funcs" subfolder. Queries are named with query number and embedded with explain (i.e., the answer to the first query in the question 3 is named "q2_query1()", and so on). The code uses the same methods for query explanations as used in question 2. The query method parameter "explain=True" controls whether to return query results or execution plan. The program to get the same visualize plan can be run from the root folder by running the "q3.py" file. The plan is saved for each query in a text file in the root folder at successful "q3.py" execution (e.g., for query 1 the file is "q3_1_plan.txt"). The output is also displayed on the console.

Query 1. Names of the top 10 most popular badges earned by users within a year of creating their accounts.

The execution plan for this query shows that database scans entire collection (forward direction column scan COLLSCAN) by unwinding badges to convert each badge as a document. Then, the match stages applies the filter to get only the badges earned with one year (in milliseconds) of the user's account creation date. The query then groups the results by badge name by counting the badge names. The sorting is applied in descending order on the count of badges and the results are restricted to 10 documents only. Finally, the records are returned with badge names only.

Query 2. Display names of users who have never posted but have a reputation greater than 1,000.

This query looks at users with reputation of greater than 1000. The query plan shows that the database will scan all documents. The query plan shows relationship with posts collection. It uses a forward filter on reputation using the column scan. Then it uses the lookup using nested loop join to map local field (user posts) with users' collection. The results are stored in user posts array. After that, another lookup filter extracts records whose users' posts are empty. Since the match is empty, the following stages do not have any documents to process.

Query 3. Display the name and reputation of users who have answered more than one question with the tag “postgresql”.

This query plan shows to include only the posts where the post type is equal to 2 (answers). With the tag “postgresql”. The query results uses the forward collection scan (COLLSCAN). Then, the plan shows grouping based on the filtered results and counting the answers. This is done by mapping the users’ id with the filtered record by joining records with users collection. Another filter removes the records with count lower than 1. The results are stored in an array called userDetails. Then, the query uses unwind to convert user details into individual documents to count answers by user. Finally, the user display names are projected from the filtered records.

Query 4. Display the names of users who posted comments with a score greater than 10 within the first week of creating their accounts.

The query fetches the names of users with comments having score greater than 10. The execution plan first filters the posts to include the posts with comments score greater than 10 using the forward collection scan. Then, the plan uses unwind to comments array to create document for each comment for each post. The following stage create user details array by joining the comments users with users collection. Another unwind operation flattens the user details by having each user with comments. Then, the execution matches the comments creation date with users’ creation date (by adding 1 week in milliseconds). Finally, the results are projected with user display names.

Query 5. The tag names of the tags most commonly used on posts along with the tag “postgresql” and the count of each tag.

The query plan uses a forward filter collection scan posts by filtering tags containing “postgresql”. Then the query unwinds the filtered collection to flatten tags array creating documents for each tag with posts. The results contains each entry with a tag and its post details. The results are then grouped by counting the tags across matching posts. Finally, the documents are sorted in descending order to view mostly commonly used tags display first.

Question 4: Taking the previous queries into account, create appropriate indexes where they are required and document your decisions. Analyze the performance differences and show your times with and without indexes.

The code for creating indexes can be run from “q4.py” from the root directory. The queries to create and list indexes are provided as “q4_index_creation()” method in “queries.py” in the “funcs” subfolder. To observe the difference in query performance and query plan generation, execute the “q2.py” and “q3.py” again.

Decisions: First, I experimented with creating indexes on all the attributes used in the queries. Then, I removed each index one by one to observe the effect on the query performance. For instance, indexes on user display name, badges name did not show any gains on the query performance. Likewise, for posts, indexes on post type and score did not add substantial value to query performance. I observed that following indexes were useful to improve the query performance.

- User Creation Date
- Badges Date
- Post Owner User
- Post Tags
- Post Creation Date
- Post Comments User
- Post Comments Score

I executed the queries and query plans again. There is a slight improvement in processing time and query plans also show a decrease and adaptation to using different methods (for some queries).

Query Performance

Table 1. Comparison of each query's execution time before and after applying indexes.

Sr	**Before (Time)**	**After (Time)**
Query 1	4.07 seconds	4.02 seconds
Query 2	0.51 seconds	0.47 seconds
Query 3	0.27 seconds	**0.015 seconds**
Query 4	0.74 seconds	**0.36 seconds**
Query 5	0.60 seconds	**0.34 seconds**

Explanations:

Query 1. Query 1 did not improve after adding indexes on used attributes as explained above. Therefore, the execution time and query plans remain almost similar.

Query 2. Query 2 shows slight improvement. The execution plan shows changes from nested loop join to use indexed loop join to map users and posts.

Query 3. This query shows significant gain in execution time. The query plan shows the matching is applied differently to check filter on tags using the index.

Query 4. This query also shows improvement in the execution time. The execution plan shows that comments score filter uses the index scan.

Query 5. This query also shows improvement in the execution time. The execution plan shows that filtering on tags is performed using the tags index.

Therefore, some improvement is overall seen with creating indexes. Some indexes like badge names and user display name do no add significant value to execution time. Therefore, only the useful indexes are added. Hence, monitoring the performance is necessary to check the applicability of each index.

Note: Instructions to run the code are provided in the README.md file in the root folder.