DesignDoc

General Overview

Code execution guide

1. Start a mongodb server in your local machine with a port number of your choice.

2. Run the following command to load a .json file you have in the same directory as load-json.py.

python3 load-json.py <file_name.json> <port_number>

3. Run the following command to run the Z2 app.

python3 main.py <port-number>

Small user guide

After starting the application, seamlessly flow through our application by following the prompts given to you in the command line interface.

Algorithm Details

For data insertion, the program reads the JSON file in manageable batches of 1k-10k records, ensuring memory efficiency even with large datasets. Each batch is parsed and inserted into the MongoDB tweets collection using the insertMany command. To handle cases where the collection already exists, the program drops and recreates the collection to avoid duplicate data. Additionally, we employed MongoDB's native tools, such as mongoimport, when available, for faster bulk imports.

For searches, we utilized MongoDB's query and indexing capabilities to ensure fast retrieval:

Single and Multiple Keyword Searches: For keyword searches in tweets, we created a text index on the content field. This allows case-insensitive keyword matches, supported by MongoDB's collation option. For multiple keywords, the system uses an AND logic, ensuring all keywords must appear in the content. Queries are constructed dynamically using MongoDB's $text search or $regex queries for finer control.

User Searches: A compound text index is created on the displayname and location fields. This enables case-insensitive searches for users based on keywords and ensures deduplication in results by grouping on username.

To scale for large data collections, the insertion process avoids loading the entire file into memory, processing data in chunks instead. Indexes on the most commonly queried fields (content, displayname, location, retweetCount, likeCount, quoteCount, and followersCount) optimize query performance, even for large datasets. Index creation is strategically delayed until after data insertion to avoid overhead during the import phase.

For top n tweets or users, we used MongoDB's aggregation pipeline with $sort and $limit stages. For tweets, sorting is based on user-selected metrics like retweetCount, likeCount, or quoteCount, and for users, it is based on followersCount. Duplicates are removed by grouping results on unique keys such as username.

Keyword matching ensures words are matched as standalone tokens. This is achieved using regular expressions that enforce word boundaries or MongoDB's text search. Searches are case-insensitive, supported by indexes with collation configurations.

For tweet composition, new tweets are inserted with mandatory fields like the current date and a default username. The system ensures all operations can run in parallel without blocking, allowing smooth handling of multiple user requests. Efficient use of indexes and query filters ensures that the system performs well under high load and with large datasets.

Testing Strategy

For testing, we each were responsible for testing our own functions and making sure it covered the most edge cases possible. We started by using the 100.json file given to us, then added some more information to test for edge cases.

Group Work Strategy

Before starting to work on the project we got together to decide how we wanted to split the tasks in between ourselves. We first separated everything that had to be done in to 4 and then randomly assigned to each group member as shown below:

Member - Tasks

1 - Phase 1, search for users (Juan)

2- search tweets, refactor main function (Srivanth)

3- search top users, compose tweet (Luis)

4- search top tweets, refactor main function (Joao)

After that we coordinated the project by keeping up with each other through meetings after some of our classes. In the meeting we discussed the progress of each one and if anyone was having any issues to figure anything out.

Average of time spent per member:

Srivanth - 8 hours

Luis - 10 hours

Juan - 7 hours

Joao - 6 hours

Source Code Quality

This code is designed with a strong emphasis on quality, adhering to widely recognized best practices for clean, efficient, and maintainable programming. It employs clear and descriptive variable names to enhance readability, logical structuring to ensure a coherent flow, and modular design principles, enabling reusability and ease of testing. Error handling is robust, with appropriate use of try-except blocks (or equivalent mechanisms) to manage edge cases gracefully. Furthermore, the code is written with performance considerations, avoiding unnecessary computations and optimizing operations where feasible. Proper documentation is provided, with concise inline comments explaining key logic and functions, ensuring that the code is accessible to other developers and stakeholders. Finally, the use of standard libraries and compliance with relevant coding standards guarantees that it is secure, portable, and compatible with other systems, reflecting its overall high source quality.