Tweet Search Application Report

Team number: 12

Team Members: Devyani Mardia, Xinran Zhao, Catherine Zhou, Hongyu Xin

GitHub repository:
https://github.com/arthurxin/Tweet-database-project

GitHub usernames for the team members:
Devyani Mardia: devyanimardia
Xinran Zhao: zhxyn6
Catherine Zhou: CathXinrouZhou
Hongyu Xin: arthurxin

Abstract

The project is trying to store and retrieve tweet data. The raw data is decoded to store in different databases and tables. MongoDB is used to store the tweet data, while SparkSQL is used to store the user information. A search engine combined with MongoDB and SparkSQL query functions is deployed as a web service. A cache database is deployed on the client to accelerate the repeat search.

1. Introduction

The architecture of this project primarily focuses on facilitating faster tweet retrieval, given the massive data set with 101,916 tweets. Each row/tweet in the dataset consisted of various components such as type of tweet (tweet, retweet, quoted tweet, multiple replies to a tweet/ replies to the replies of a tweet, or a combination of the retweeted and quoted tweet.), user information: (user name, screen name, followers, favorite count, etc.) and post information (favorite count, retweet count, reply count, and quote_count). The project aimed to separate tweet storage in our Nosql type data store and user information into the SparkSQL data source for easy information retrieval. As SparkSQL is excellent for structured data, the first point of segregation was to use SparkSQL to store all user information with around 90336 unique users. The NoSQL data store allows storing data as a document/JSON. It gives the freedom to accommodate different data types without pre-defining a schema. Hence Mongo db was used to provide adaptability and flexibility in storing the tweet information. The main components of this project are SparkSQL data storage and retrieval, MongoDB data storage and retrieval, Caching, and the main search application to leverage the different enhancements done in retrieving a tweet.

2. Framework and Deployment Setting

The project is deployed on three virtual machines on the Google Cloud Platform. A cluster of MongoDB and SparkSQL is installed on the three virtual machines. Once the storage or computation power is insufficient, a new virtual machine could be added to the cluster. The search engine would be deployed on the server by socket or flask. Clients could be deployed on any platform(e.g., website, IOS) and use any programming language.

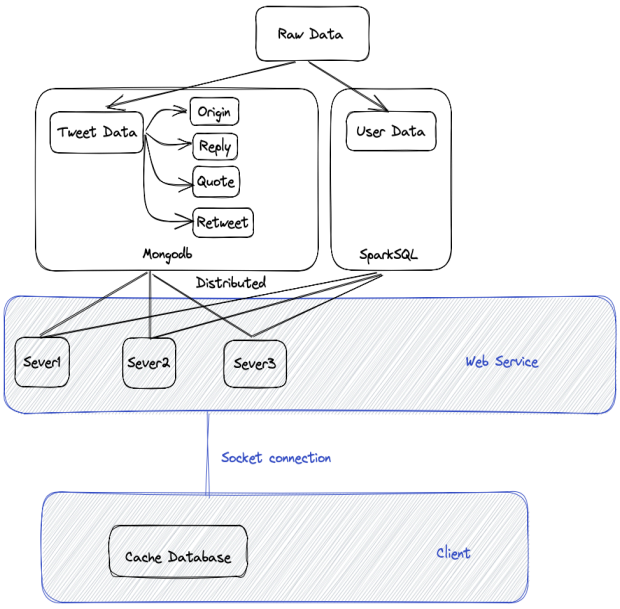

Figure 1 Framework of Project

3 Data Decoding and Insertion

3.1 Dataset Description and Decoding

The dataset features a nested structure, including original tweets along with related retweet and quote tweet data. Each tweet may contain retweet or quote tweet data, with retweets and quotes embedded throughout the data structure, creating multiple layers. Related user information is present at each level.

The recursive function solved the nested structure perfectly. If the tweet has no nested structure, the tweet will update the database. If the tweet has a nested structure, the nested tweet will be recursively processed first and then updated.

```
def recursive_update(input_tweet):
    if input_tweet have no nested structure(reply or origin):
        update the  input_tweet in database if [timestamp] is larger
    If input_tweet have nested structure(quote or retweet):
        recursive_update(input_tweet[nested_tweet])
        update the input_tweet in database if [timestamp] is larger
```

Figure 2 Pseudocode of recursive update

3.2 Tables on MongoDB

3.2.1 Description of tables on MongoDB

The outputs of decoding the raw data led to the creation of mainly four collections on the tweet side: origin_tweet, reply_tweet, retweet_tweet, and quote_tweet. Each table stored all the tweets belonging to the same type. One tweet may be stored in two or more tables simultaneously.

The reference is added to each tweet record after the tweet is inserted in the database. That ensures each record is complete with tweet data and references. The references about the tweet to another tweet are “reply to”, “quoted to” and “retweeted to”. The reference for other tweets to the tweet is “reply list”, “quoted list” and “retweeted list”.

3.2.2 Indexes of tables on MongoDB

3.2.2.1 Text Indexes

To run text search queries, a text index creation is a must on the collection. Text indices on text and hashtags fields for

origin_tweet, reply_tweet, quote_tweet, and retweet_tweet collections. A collection can have only one text index but cover multiple fields. Hence our text and hashtags are clubbed together, which helps facilitate searching a keyword in text and hashtags simultaneously. Weights addition can help denote a field's significance relative to other indexed fields regarding text score search. Text indices help to limit the number of index scans by using compound indices, which will work in the first-in-first-out format so the first specified condition is extracted first. It will search documents only inside this subset, thus limiting our scan counts. The project had a custom score field that helped determine the tweet scores and sort them according to their popularity.

3.2.2.2 Compound Indexes

Multiple fields are referenced in this type of index. Creating indexes, it is crucial to consider the order in which these indices are created as it would help with the effectiveness in which indexes are created. Indexes inside compound indexes can be used in the prefix index style. They are the beginning subsets of indexed fields and work in the "and" condition fashion.

3.2.2.3 Sorting for search indexes

Sorting is a fundamental concept as it enables the database to improve its performance and reduce the index scans. Sorting is heavily interlinked with Text indices as it enables document scans of only the subset of documents that meets certain criteria and the next criteria in an incremental "and" fashion, thus reducing our index scans. Sorting in ascending order can be achieved using 1 and in descending order using -1 on a certain field.

3.2.3 popular score of tables on MongoDB:

The custom score is a field added to the database to understand the popularity of a tweet. This field can be calculated in several ways which would also impact the popularity of the tweets. It can be treated as a weighted sum of the four fields: favorite_count, retweet_count, reply_count, and quoted_count. The weights would determine the list of tweets getting popular. The general thumb rule is that Retweets and replies are given more priority over quoted and favorite counts. However, the significance is not heavily skewed. Hence to keep things discrete and simple for the project.

There can be other techniques where a combination score can also be created to give

the popularity of a tweet based on nested retweet/ reply popularity and calculate the overall weighted score that a popular tweets has created or, in simple words, how many impressions were created by this post to get the entire picture/impact of this tweet.

3.3 Tables on SparkSQL:

Two tables have been created for the user information: the *user_basic* table and the *user_profile* table. The user_basic table is mainly used for search.

3.3.1 Insert with data frame SparkSQL

Recursively processing each line of data in the file, all user information within each line is stored in a data frame. After processing all rows in the file, the data in the data frame is inserted into the table in a single step. Before adopting this insertion method, attempts were made to insert user data into the table as it was encountered. It proved very time-consuming, taking about an hour to store a thousand rows of user data. Subsequently, it was discovered that Parquet in SparkSQL is a columnar storage format very well suited for big data processing, a binary file format designed for efficiently storing large amounts of structured data. As a result, this storage format was chosen, leading to the initially mentioned approach.

3.3.2 Aggregate by user ID

Now, there is a table containing all user information, but numerous duplicate users take up extra space and cause ambiguity. As the next step, the data is aggregated based on the user ID, which serves as the primary key in the new table. All tweet IDs related to a particular user are stored in a new column as a list. The most recent versions of influence-related data, such as the number of followers and likes, are retained. Notably, a timestamp has been stored from the beginning, representing the update time for user data. This timestamp enables comparison to determine the latest user data.

4 Searching engine

The search engine class initials with connections to MongoDB and SparkSQL. Searching request is designed to be <searching type> + <searching config> + <searching keywords>. Different searching configs with the same search keywords could avoid the same return from the cache. First, the searching request will be sent to cache. If no result is returned from cache, the searching engine will go to the server query MongoDB or SparkSQL.

Using the different functions to smoothen our search also raises the topic of what type

of the search is needed. For example, it can be a text string search, hashtags searching, exact search text, stop word text search, fuzzy text search, etc. All these are important to user style searching and can be achieved by using text indices to a great extent.

4.2 search on the MongoDB

MongoDB search is a basis for the future search algorithm. There are two primary types of searches: tweet id search and hashtag/keyword search.

Tweet id search: this search can retrieve all the tweet information by their unique tweet id. To execute the tweet id search, the regular expression is created to find the particular tweet id, which is then used as the search query by the find method.

Hashtag/Keyword search: this search can return all the needed information that contains a specific hashtag or keyword. To execute the hashtag/keyword search, the "$search" condition is used to query using the find method.

This gives a basic idea and understanding of the search algorithm. To make the search executable for the entire MongoDB database, functions are defined, and four cases are considered, each corresponding to one of the four collections in the database.

After connecting to MongoDB, the algorithm can be implemented by locating the specific collection containing the data of interest. Once the collection is located, the find method is used to search for the particular tweet id in that collection. This can be applied to all four collections.

In the case of the reply_tweet collection, there may be comments that are replies to previously retrieved replies. To handle this situation, a recursive function is defined to track these comments. This allows users to call the function recursively to check for additional replies to the original reply. This enabled a user to check the comments to a tweet and the comments to a particular comment as well. On searching a tweet a user got a tweet' replies as well as all the nested replies to this reply and so on.

Users could retrieve the top 10 or n tweets of all time using the top_10 function. This majorly focused on using custom_score to get the users the top tweets of all time to get the user engaged and interested in joining/using the application

Advanced Search is another functionality to enable user search to search a keyword in Text as well as replies in case the search results didn't have enough records and a user, may be interested in checking the related comments/ replies to a related topic

Based on previous keyword/hashtag search, the general function can be defined. The sorting method is considered, which is timestamp by default. The number of entries is set to 10 by default. Since the origin_tweet collection contains all the necessary data, the search query will be based on this collection. According to the previous structure, using an empty list to append all the data needed to the answer list. This will ensure that all the relevant data is captured and included in the search results. This can apply to the replies keyword search by changing the origin_tweet to reply_tweet collection.

4.3 search on the SparkSQL

When searching for usernames, SparkSQL search comes into play. As a search string is an input, the search engine will look for matching users within the user basic information table, specifically in both the name and screen_name columns. Subsequently, the user's identity information(including *user_id, name*, etc.), influence (including the number of followers, likes, etc.), and a list of all tweet IDs related to the user will be returned as output.

There are two matching patterns employed: fuzzy matching and exact matching.
Fuzzy matching can return a broader range of results, allowing users to find potentially relevant matches even if their search string is not an exact match. This approach is advantageous when users need help remembering the precise spelling or wording of a name or screen_name. However, the downside of fuzzy matching is that it may return too many results, some of which may need to be more relevant to the user's search intent.

On the other hand, exact matching is more strict and only returns results that match the search string precisely. But, the drawback is its low tolerance for errors incorrect characters in the search string will prevent matching the target user.

4.3 Cache Database

The cache database consists of a dictionary and a min-heap. The dictionary's key is the

searching request, and the value is ([list of searching timestamps], content). The key of the heap is the timestamp of the request sent, and the value is the query. For achieving O(1) computation complexity, the dictionary is the only best choice. However, the dictionary data structure can't save information about the insertion order. To implement the feature of deleting the oldest record automatically, a min-heap is used to save the insertion order. The root node of the heap is the oldest searching query. Insertion and deletion operations would take O(n) = log(n). Once the heap size reaches the limit set, a batch of the oldest record will be deleted, and the persistent system will write the heap and dictionary into the disk.

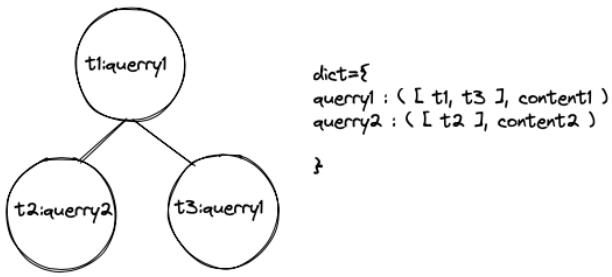

Figure 3 example of three insertions on the cache database

5. Performance

The text index greatly improved the speed of searching in the mongo db. Using regular expressions to implement the keyword searching feature would take 1.97s to get 100 lines of records. Benefiting from the text index and text search function supported by MongoDB, keyword searching is accelerated, it just takes about 290ms to get 100 lines.

Hashtag searching searches on the hashtag fields. It only cost 276ms to get 100 lines return. The slowest searching type is the user searching, it costs 330 ms to get 100 lines return. The reason is that indexes cannot be added to SparkSQL easily. If the query and content are stored in the cache database, it only takes about 1 Millisecond to hit the record.

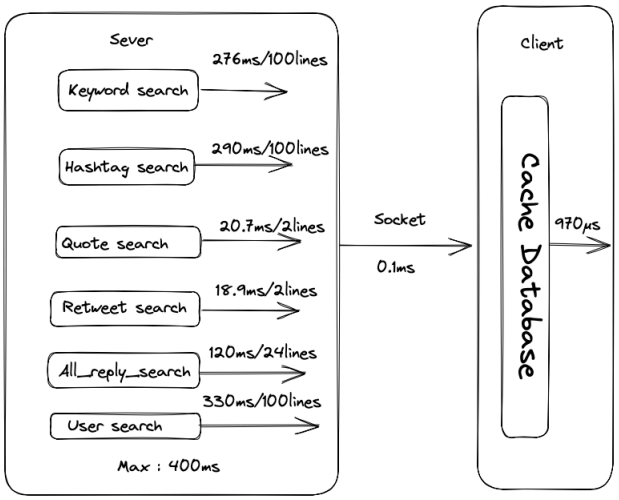

Figure 4 time cost of searchings

6. Client

The Searching engine is deployed as a flask service on the server. Each search function is deployed as a website. For example, keyword searching is "ServerIP:port/keyword_searching." . Clients pass a search request to the flask server to get the search result. The Cache database is deployed on the Client side. Importing the client tweet search class could support all features of the project.

7. Conclusion:

The highlight of this project is that utilizing core database concepts like Caching, Indexing, scoring, and segregation(main tweets collection into multiple subcomponents: original tweet, retweet, quoted tweet, reply tweets)/ database schema design showed an optimum way to store and query data to serve customers in the most optimal way.

The MongoDB and the SparkSQL should be deployed on different virtual machines. And those machines could be customized for the database. If they are deployed on the same virtual machine, thread, and memory control are necessary to avoid out-of-memory.

SparkSQL is good at table-level operation. For Searching one query, some other database could be better.

8. Future Scope:

A more complicated cache database could be deployed on the server using Spark RDD. Client-level cache database solved the repeat searching from one user. The server-level cache database could solve the frequent searching from different clients.

9. Reference

Project URL:

https://github.com/arthurxin/Tweet-database-project

MongoDB. (n.d.). *MongoDB: The Developer Data Platform*. https://www.mongodb.com/

Spark SQL & DataFrames | Apache Spark. (n.d.). https://spark.apache.org/sql/

JSON Formatter & Validator. (n.d.). JSON Formatter &Amp; Validator. https://jsonformatter.curiousconcept.com/#

The Python Standard Library. (n.d.). Python Documentation. https://docs.python.org/3/library/

Twitter API Documentation. (n.d.). Docs | Twitter Developer Platform. https://developer.twitter.com/en/docs/twitter-api

10. Project Responsibilities

Data Query/Insertion SparkSQL - Xinran Zhao

Data Query MongoDB - Catherine Zhou

Search Algorithm MongoDB - Devyani Mardia

Platform Deploy, Caching - Hongyu Xin