Building a Comprehensive Twitter Database: Using MySQL and ElasticSearch for Searching Application

TEAM 15
Yunhao Li [*EmmerichLi*][*], Siheng Huang [*SIHENG-H*],
Yanhan Chen [*Hinn-Chen*], Yicong Li [*Sonisli*]
GitHub: `https://github.com/SIHENG-H/RU_DataBase_Management_694_2023_team15`

May 2023

1 Introduction

The original tweet data was collated and processed, followed by a reconstruction of the retweet data structure. Three relational data stores were established using MySQL to store basic tweet information, user information, and tweet popularity. A non-relational data store was built with ElasticSearch to store the text of all tweets. A cache was set up to store the most popular user data, saving searching time by avoiding the need to pull data from the database repeatedly. A search application was built upon these three components, offering functions such as searching for tweets, users, and the most popular tweets and users.

2 Dataset

The utilized dataset is 'corona-out-3', containing information from 101,894 tweets. This information is stored in a dictionary-like format, which includes data about each tweet such as the posting time, tweet ID, text, source, and user information like user ID, name, location, URL, and creation time. Interactive messages, such as quotes, replies, retweets, and favorites, are also included. A sub-dictionary records the retweets of each tweet; if a tweet is a retweet, the dictionary value comprises all aforementioned information of the retweeted tweet. Some additional information, not relevant to the search application, is also present but will not be discussed in detail.

3 Persisted Data Model and Datastores

3.1 MySQL

MySQL, an open-source relational database management system (RDBMS), is widely employed for web applications, providing a versatile and efficient means of data storage and retrieval. To store information about Twitter, three relational database tables were created using MySQL.

[*]The contents in parentheses are the respective GitHub names

The first table stores basic tweet information, including ID, user ID, and creation time, along with the ID of the retweeted tweet (if applicable) and the IDs of tweets that retweeted it (if applicable). In this table, the primary key is the ID.

The second table holds user information, such as user ID, user name, location, URL, and user popularity. In this table, User ID serves as the primary key and acts as the foreign key of the first table.

The third table focuses on tweet popularity, which is evaluated using quote count, reply count, retweet count, and favorite count. Although a more sophisticated algorithm could be applied for calculating popularity, the project's main goal did not emphasize this aspect, so the four numbers were simply summed. The higher the sum, the more popular the tweet. In this table, ID serves as the primary key and also functions as the foreign key of the first table.

3.2 ElasticSearch

Elasticsearch, a schema-less, JSON-based search engine, is optimized for full-text search and high availability, employing a cluster-based architecture with horizontal scalability. On the other hand, relational databases utilize a schema-based model with tables and SQL for data manipulation, providing ACID transactions for data consistency.

Initially, the Elasticsearch server was connected, and a temporary node was set up. Subsequently, a database with the index 'tweets' was established. Notably, only the ID and text of the tweet are stored in Elasticsearch to avoid duplicate content and save space, as the same information is already stored in MySQL.

4 Processing tweets for storing in datastores

4.1 Processing Retweet

Considering a tweet and its retweets as a tree, the existing data provides a way to identify the parent node of any given node. This is sufficient to acquire the complete retweet relationship. However, the data structure may not be intuitive. To find all retweets of a tweet, or all child nodes of any node, the entire dataset must be traversed. The time complexity of this behavior is O(n), which can be expensive when dealing with large amounts of data in the database.

To address this issue, a key-value pair was added for each tweet in the original dictionary. The value is a list of all retweets corresponding to that tweet. For example, if A is a tweet and B and C are its retweets, the value of A is a list containing the IDs of B and C. If a tweet has no retweets, the value will be an empty list.

Completing the aforementioned operations requires only O(n) time to traverse the dataset once during data loading. When a new tweet is added to the database, it takes O(1) time to add it to the corresponding list of the retweeted tweet.

Additionally, the information in 'retweet_status' of all tweets was extracted. These tweets were not incorporated in the previously mentioned dataset. In total, there are 14,842 such tweets.

4.2 Loading Data - MySQL

Before loading the data, a new dictionary was created to store useful features, discarding any irrelevant information.

Subsequently, the data was loaded into MySQL's database one at a time using the following code.

```
for key, val in pro_tweets.items():
    cursor = mydb.cursor()
    query = "INSERT INTO tweet_information (tweet_id, user_id, created_at, retweet, retweeted) VALUES (%s, %s, %s, %s, %s)"
    param = (val['id_str'], val['user_id_str'], val['created_at'], json.dumps(val['retweet']), val['retweeted'])
    cursor.execute(query, param)
    mydb.commit()
```

Listing 1: Loading data to database 'tweet_information'

The aforementioned code loads the data into the first database, 'tweet information'. The code for the remaining two databases follows a similar structure.

	tweet_id	user_id	created_at	retweet	retweeted
0	1210899520692740096	1133262236745588737	Sat Dec 28 12:25:17 +0000 2019	["1254022996232028161"]	
1	1217113384291880962	61519237	Tue Jan 14 15:56:58 +0000 2020	["1254050769202032642"]	
2	1221138283675021312	29280466	Sat Jan 25 18:30:29 +0000 2020	["1254053174052360192"]	
3	1221341690142679041	349747452	Sun Jan 26 07:58:45 +0000 2020	["1254039171905150978", "1254039540383145990"]	
4	1222318351357054976	743662396892282881	Wed Jan 29 00:39:39 +0000 2020	["1254039762962460672"]	

Figure 1: Database 'tweet_information'

	user_id	name	location	url	user_popularity
0	1000006582896295938	sara	None	None	0
1	1000026406112251905	Corona supplier banker	Hell	None	0
2	1000027886915637250	cheche	None	None	0
3	1000034375973646337	clarih 🐻	Bonsucesso	None	0
4	100004211	azakiya tamilmagan	Chennai, India	None	0

Figure 2: Database 'user_information'

	tweet_id	quote_count	reply_count	retweet_count	favourite_count	popularity
0	1210899520692740096	0	1	3	2	6
1	1217113384291880962	0	2	15	70	87
2	1221138283675021312	5	23	306	797	1131
3	1221341690142679041	68	83	292	286	729
4	1222318351357054976	1072	2719	11983	61466	77240

Figure 3: Database 'tweet_popularity'

The three displayed graphs 1 2 3 represent partial data from the three databases. Notably, the 'retweet' field in the first table, as mentioned earlier, consists of a list of all IDs that have been retweeted for the corresponding tweet.

4.3 Loading Data - ElasticSearch

In contrast to loading data into MySQL, data is not loaded one by one in Elasticsearch. Instead, the helper package in Elasticsearch is utilized. As demonstrated by actual testing, the time required for batch loading is only one-tenth of that for individual loading. The code used for this process is provided below.

```
actions = [
    {
        "_index": 'tweets',
        "_id": key,
        "_source": {"text": val.get('text')},
    }
    for key, val in pro_tweets.items()
]
helpers.bulk(es, actions)
```

Listing 2: Loading data to index 'tweets'

5 Searching Application Design

5.1 Cache

The cache design is illustrated in the following diagram 4.

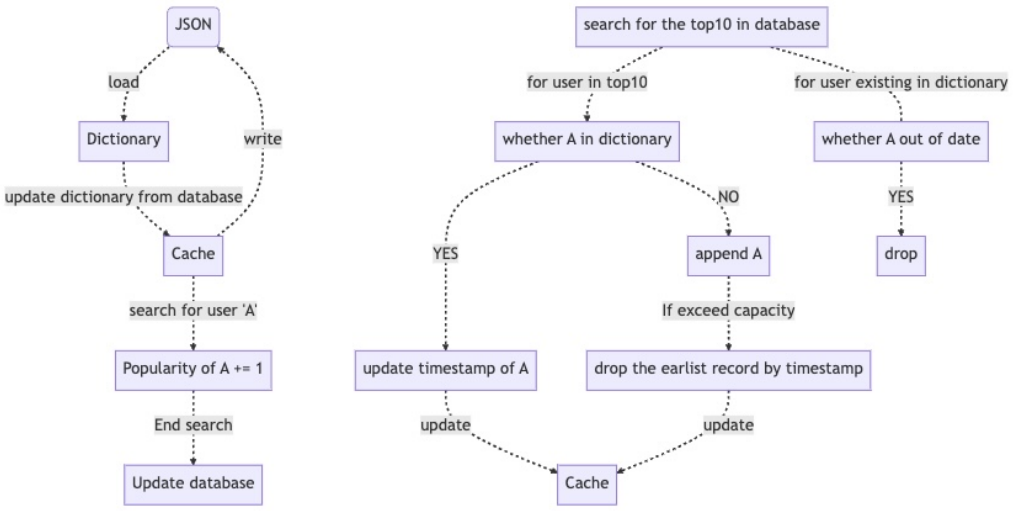

Figure 4: Flow chart of cache

The right section of the diagram provides a detailed explanation of the update process from the database.

The cache was defined to store the most popular user information. Formally, this is a dictionary, identical to the user information table in MySQL, including user ID, user name, location, URL, and popularity. Additionally, a unique feature, time to live, represents the user's timestamp.

The cache size was set to 20 and a function to check the dictionary's length was defined; if it potentially exceeds 20, certain records are excluded based on a set of rules.

With a static database, user popularity was calculated based on search operations alone. When a user's information is searched, their corresponding user popularity increases by one.

The cache is loaded each time the search application starts. Upon initially starting the application, all information of the 10 most popular users and the current timestamps are added to the cache. For each subsequent update, if a new popular user emerges, they are added to the cache. If this addition causes an overload, the record with the earliest timestamp is removed. Simultaneously, the timestamps of existing records in the cache are updated.

```
def drop_to_capacity(self):
    while len(self.dictionary) > self.capacity:
        earliest_key = min(self.dictionary, key=lambda k: (self.dictionary[k]['time'
    ]))
        del self.dictionary[earliest_key]
```

Listing 3: Removing the earlist key

Lastly, an 'out of date' rule was established. If a record's timestamp in the cache is from 3 days prior, the record is removed.

```
def drop_out_of_date(self):
    time_now = datetime.now()
    three_days = timedelta(days = 3)
    for key in self.dictionary:
        if time_now - self.dictionary[key]['time'] > three_days:
            del self.dictionary[key]
```

Listing 4: Removing out-of-date key

5.2 Initialization of Searching Application

At the start of the application, three objects are instantiated from each of the three classes.

Search methods are incorporated in each class. In the 'cache' class, methods to search for users directly from the cache are included. The 'ESearch' class contains fuzzy search and full-text search methods, as well as methods for searching tweet text by ID. In the 'SQL' class, various search methods are available, such as searching for tweets and user information by tweet ID, searching for tweets by user ID, searching for user information by user name, searching for top tweets and top users, and more.

The cache is then loaded from a local JSON file, which is essentially a dictionary. Next, the cache records are updated from the database. The updated cache information is subsequently written to the local file.

In the app, users can search for tweets and user information. Additionally, fixed search options are provided, such as the top 10 most popular tweets and the top 10 most popular users.

The content of this section is provided by the following code:

```
from Cache import Cache
from ES import ESearch
```

```
from MySQL import Sql

if __name__ == '__main__':
    cache = Cache(20)
    sql = Sql()
    es = ESearch()

    cache.update()
    cache.write_to_file()

    print('Welcome to the search application designed by Team 15\n')
    flag = True
    while flag:
        print('Please select one of the following four options for searching\n')
        print('1. tweet')
        print('2. user')
        print('3. Top ten tweets')
        print('4. Top ten users\n')
        option = input('Please enter a single number between 1 and 4: ')
```

Listing 5: Searching application - initialization

5.3 Searching Tweet

When attempting to search for tweets, the application first checks the length of the search text. If it is a single word, fuzzy search is preferred due to its higher accuracy. If it is a phrase or sentence, full-text search is used instead. Elasticsearch automatically sorts search results in descending order of text relevance.

Users can choose to view the details of a specific tweet, including its posting time, user information, list of retweets, and more, by entering the number preceding the result. This information is fetched from the basic information table in MySQL. Notably, the list of retweets created while processing the raw data becomes useful in this context.

The search results for 'unemployment' are displayed below 5.

5.4 Searching User

When searching for user information, the app first checks the cache. If no corresponding information is found there, the system searches the user information table. If the search is successful, the app provides basic information about the user along with their five most recent tweets.

The search results for 'sara' are displayed below 6.

5.5 Searching Top10 Tweet and User

For the top 10 most popular users and tweets, lists are provided along with the corresponding drill-down search functionality. The top 10 users are obtained directly from the cache, while the top 10 tweets are retrieved from the tweet popularity table.

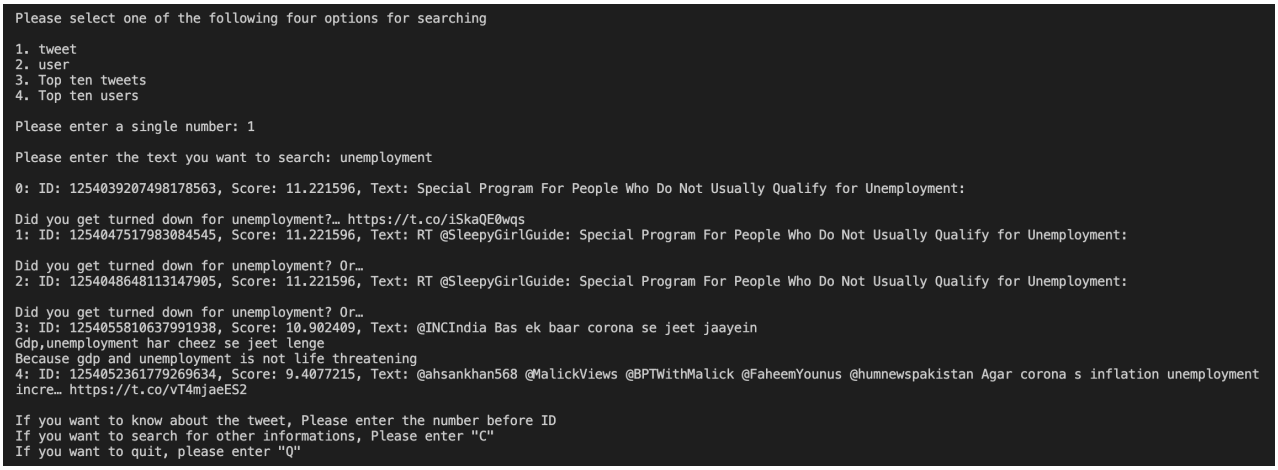

Figure 5: Search result of tweet

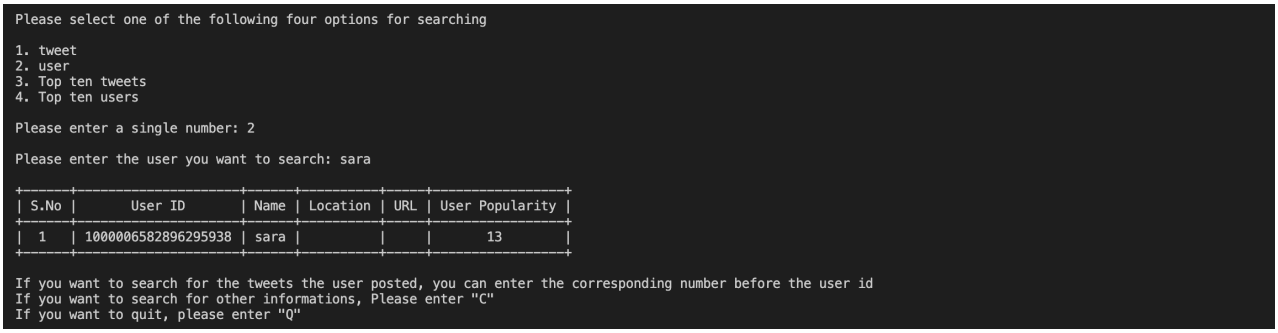

Figure 6: Search result of user

Due to space constraints, the entire picture is not displayed 7. However, the rest of it follows the same pattern. Users can search for additional information about a tweet by typing in the number preceding the tweet.

The top 10 most popular users are displayed below 8. Similarly, users can query all tweets posted by a specific user by entering the serial number preceding the user ID.

6 Results

A comparison is made between the duration of searching for user information from the cache and that from the database. The search code and the corresponding results are shown below 9 10.

```
begin = time.time()
cache.search('linda')
end = time.time()
print(f"Time taken for the operation: {end - begin:.5f} seconds")
```

Listing 6: Searching time from cache

```
begin = time.time()
sql.search_user_from_username('cheche')
end = time.time()
```

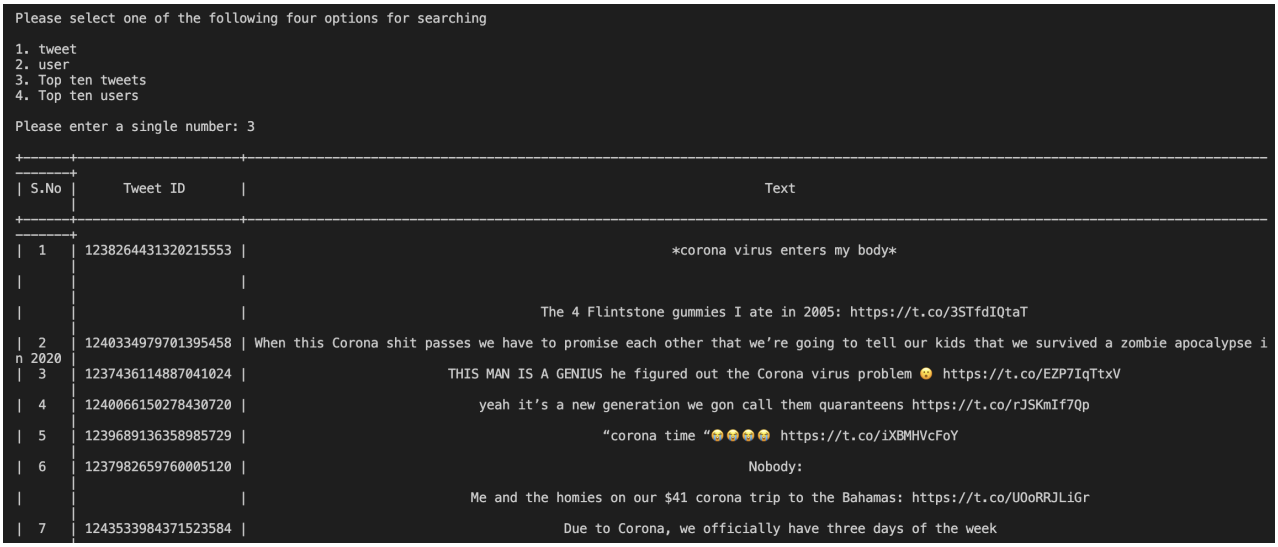

Figure 7: Search result of top10 tweet

Figure 8: Search result of top10 user

```
print(f"Time taken for the operation: {end - begin:.5f} seconds")
```

Listing 7: Searching time from database

As evident from the results, the search from the cache took 0.00054 seconds, while the search from the database took 0.05846 seconds. The difference is almost 100 times, which highlights the reason for utilizing a cache in this application.

7 Conclusions

Certainly! Here is a possible conclusion that you can use for your project report:

In conclusion, our project on MySQL, Elasticsearch, and caching has demonstrated the effectiveness of integrating these technologies to build a high-performance search application. We have shown that MySQL can handle structured data and complex queries, while Elasticsearch can search and analyze unstructured data. Caching, meanwhile, can help to improve performance and reduce

Figure 9: Search result from cache

Figure 10: Search result from database

response time.

Through the development of our search application, we have showcased the potential of these technologies when used together. Our application was able to handle both structured and unstructured data, providing fast and accurate search results. We have also demonstrated the benefits of caching, which helped to improve search speed and responsiveness.

Overall, our project has highlighted the importance of combining different technologies to build powerful and efficient applications. The integration of MySQL, Elasticsearch, and caching provides a solid foundation for building search applications that can handle complex data requirements and deliver fast and accurate results. We believe that our project has contributed to the wider understanding of how these technologies can be used together to create effective search applications.

Roles and Responsibilities

Siheng Huang managed the raw data and constructed the relational database in MySQL.

Yicong Li researched Elasticsearch and fed the text of tweets into an Elasticsearch index.

Yanhan Chen contributed the key inspiration for designing the cache and incorporating it into search applications.

Yunhao Li integrated the work of the first three, rewrote the code, and completed the design and packaging of the search application.

Collaboratively, the team worked on creating slides and preparing the presentation.