Database Management Project

Interim Report

Team-18

Arun Mishra, Yiming Tan , Fangru Linghu, Siyu Chen

This interim report is a briefing of our approach towards this project.

It includes 4 part as follows:

Parts of the Projects and responsible personal:

- Datastores
  store the information in the tweets in at least 2 datastores.
  - relational datastore(handled by Fangru Linghu)
  - non-relational datastore (handled by Arun Mishra)
- Cache (handled by Yiming Tan)
  Design and implement a cache for storing "popular" (frequently accessed) data.
- Search Application(handled by Siyu Chen)
  Design a search application for your tweet store. You must provide several options such as search by string, hashtag, and user at the minimum.

Github link for project repository: **Project for 694-2023-team18**

Github_usernames of the team

Fangru Linghu(relational datastore): FangruLinghu

Arun Mishra(non-relational datastore) : arunmishrarut

Yiming Tan(Cache): Led854

Siyu Chen(Search Application): Siyu-789

1 Relational Datastore

1.1 Data Description

The corona-out-3.csv file primarily contains data related to tweets about the Coronavirus. The data reflects two types of actions, tweets and retweets. It also includes information about the users who performed these actions, such as User ID, User Name, User Screen Name, User Location, User Website URL, and User Description, among others. In this search application, user information is mainly used for searching and determining user interface states. Most of the information is redundant and can be dropped directly.

1.2 MySQL

In this section, MySQL is employed to store user information.

MySQL is an open-source relational database management system (RDBMS) that has gained widespread popularity due to its performance, stability, and ease of use. MySQL is commonly used for a variety of applications, including web development, data warehousing, e-commerce, and more.

Benefits of using MySQL to store user information:

- MySQL is a relational database management system that organizes data in tables with rows and columns. This structure is well-suited for handling user information, which consists of relationships with other data entities.

- MySQL can handle a large amount of data and scale to accommodate the growth of user information over time. It can support millions of records while maintaining efficient query performance.
- MySQL offers built-in mechanisms, such as primary keys, foreign keys, and transactions, to ensure data consistency and integrity.

1.3 Schema

Based on the search application’s features, the four properties below is chosen from user imformation:

id_str **(Primary key)**	String	The user's unique identifier. In the context of Twitter's database, the user ID uniquely represents a user.
name	String	The user's real name. This field is not unique within Twitter's database, meaning multiple users may have the same name.
screen_name	String	The user's username. It is a unique and recognizable string used to represent the user on Twitter. No two users can have the same username.
protected	Boolean	Describe whether the account is protected or not. When the “protected” value is set to “True”, only approved followers can view the user’s tweets and retweets. When the value is set to “False”, it means that the user’s tweets and retweets are visible to everyone.

The first three: *id*, *name* and *screen_name* are used in searching users. And *protected* is used in identifying user interface states.

1.3 Optimization

Optimizing data storage formats to enhance performance.

1.3.1 Dropping irrelevant fields

Removing fields that are not required for the search application to reduce storage space and minimize the amount of data when processing search queries.

As shown above, only four fields are retained in this application.

1.3.2 Indexing

Creating indexes on tables based on query requirements can significantly increase query speed. It is important to carefully analyze the most common query patterns and identify the columns that would benefit most from indexing.

1.3.3 Trade-off

Optimizing data storage comes with certain trade-offs:

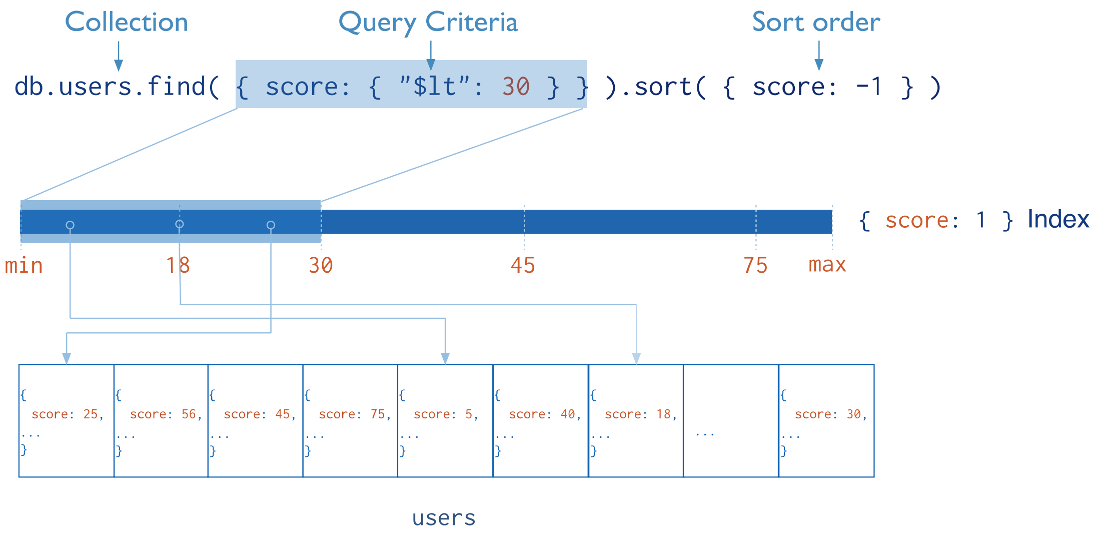
- Storage space: Indexes consume additional storage space. As the number of indexes increases, so does the storage requirement for maintaining those indexes.
- Insertion/Update performance: When inserting or updating data, the presence of indexes might decrease performance, as the index structures need to be updated as well. The more indexes there are, the more overhead is involved in keeping them up to date during data modifications.
- it is important to carefully evaluate which fields are truly irrelevant before dropping them.

2.Non-Relational Data storage

The structure of a collected tweet can be quite complex and thus a non-relational DBMS is used to store these tweets. Considering the fact that tweets are like small transaction type data(data size is not that large per tweet) that is OLTP type DBMS will be suited.

Therefore, MongoDB is used to store non-relational data. MongoDB is known for fast querying capabilities and automatic scalability. Data is stores in JSON-like documents with dynamic schema which further justify its usages to store tweets, since tweets data is in JSON.

Indexing is a crucial part of data storage and efficient retrieval/querying. In MongoDB,indexes are tree-structured sets of references to documents. The query planner can employ indexes to efficiently enumerate and sort matching documents. Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection. The following diagram illustrates a query that selects and orders the matching documents using an index:



```
db.users.find( { score: { "$lt": 30 } } ).sort( { score: -1 } )
```

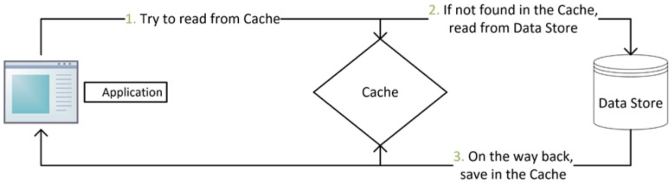
3. Cache

Caching is a crucial component of the layered architecture of the search application, typically used to improve user experience by reducing waiting time, and reduce server workload.

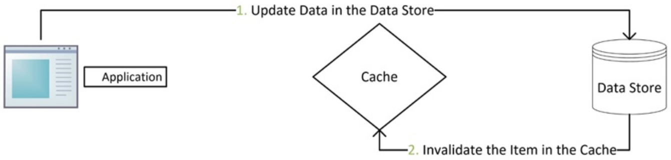
Redis' cache service is selected for caching in this project. As Redis delivers high throughput and low-latency access to data, and the fastest possible performance when reading or writing data.

Cache-Aside Pattern (Lazy caching) is adopted as cache mode, as the cache contains only data that the application requests, which helps keep the cache size cost-effective. However, some overhead is added to the initial response time because additional roundtrips to the cache and database are needed.

The reading procedure is showed as below:


1. Try to read from Cache
2. If not found in the Cache, read from Data Store
Application
Cache
Data Store
3. On the way back, save in the Cache

The order for writing is swapped based on the default process (attached below) in this project for better consistency. Nevertheless, this order may lead old data into the cache. A delayed double-deletion strategy has been used to fix this scenario.


1. Update Data in the Data Store
Application
Cache
Data Store
2. Invalidate the Item in the Cache

Some of the other considerations include applying time-to-live (TTL) field to all of the cache keys to handle stale data, and evicting the least-accessed keys by default as the cache fills up.

Additional optimization methods like dimensional caching will also be explored in caching, as there exists slowly changing dimensions such as user info, simply updating the dimensions instead of updating the whole data would be a good way to improve performance.

4. Search Application

4. Searching Application

This section aims to present the design process of building a search application for this tweet store using DuckDB, which is an embedded SQL OLAP database management system that supports various types of searches and drill-downs, as it allows executing standard SQL queries.

4.1 Search Features and Drill-downs Capability

The application supports the following search types:

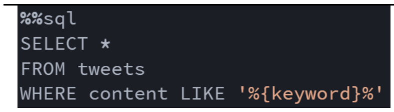
4.1.1 Text Searches

Searching by string: Allows users to search for tweets containing a specific string.

Searching by hashtag: Allows users to search for tweets containing a specific hashtag.

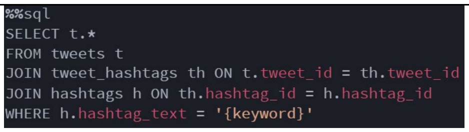
Searching by user: Allows users to search for tweets from a specific user.

PICTURE 3.1.1-1 Query String Searching



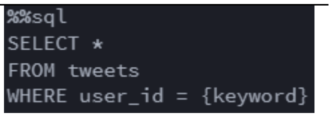
```
%%sql
SELECT *
FROM tweets
WHERE content LIKE '%{keyword}%'
```

PICTURE 3.1.1-2 Query Hashtag Searching



```
%%sql
SELECT t.*
FROM tweets t
JOIN tweet_hashtags th ON t.tweet_id = th.tweet_id
JOIN hashtags h ON th.hashtag_id = h.hashtag_id
WHERE h.hashtag_text = '{keyword}'
```

PICTURE 3.1.1-3 Query User Searching



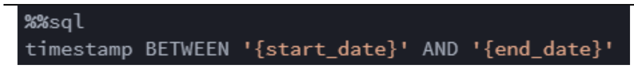
```
%%sql
SELECT *
FROM tweets
WHERE user_id = {keyword}
```

4.1.2 Advanced Searches

Searching limited by time: Allows for a time range in text searches.

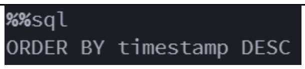
Ranking: search results can be ordered based on tweet timestamps.

PICTURE 3.1.2-1 Query Timestamp Limitation



```
%%sql
timestamp BETWEEN '{start_date}' AND '{end_date}'
```

PICTURE 3.1.2-2 Query Timestamp Order

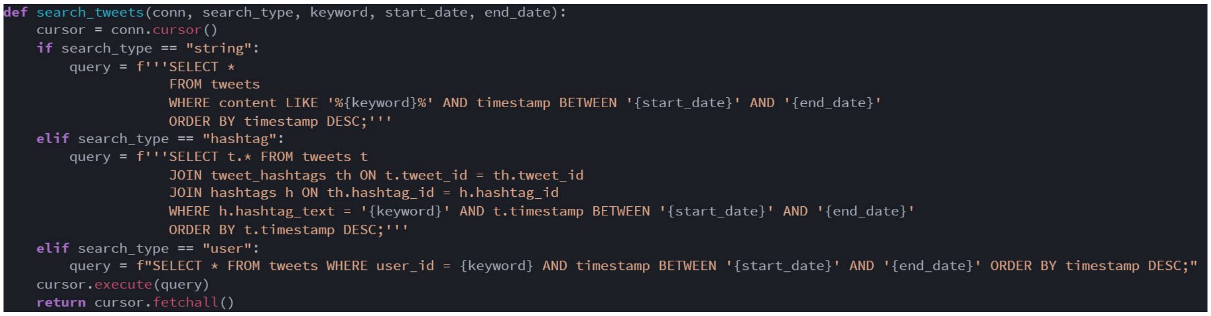


```
%%sql
ORDER BY timestamp DESC
```

4.1.3 Drill-Downs And Aggregation

Drill-down features include displaying metadata of tweets (such as author, time of tweet, and number of retweets) and viewing other tweets or retweets by a specific author. By using appropriate clauses and operators in SQL queries, we translated search queries into queries for data storage.

4.1.4 Searching Application

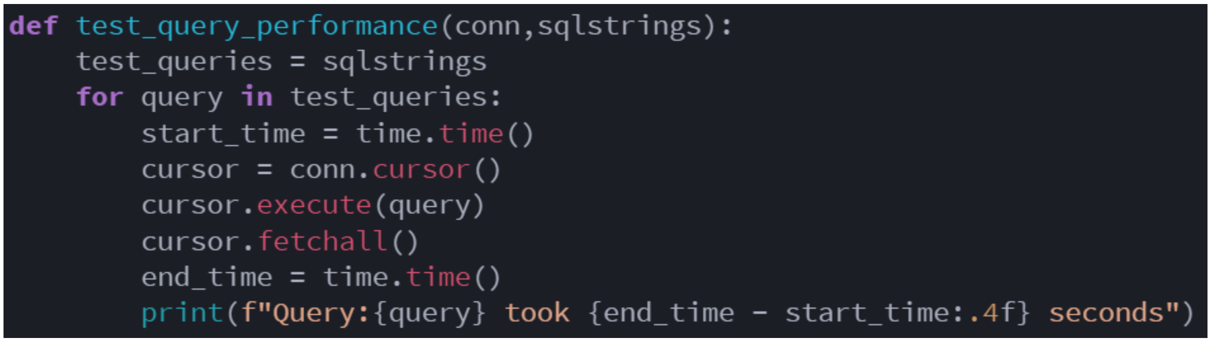


```
def search_tweets(conn, search_type, keyword, start_date, end_date):
    cursor = conn.cursor()
    if search_type == "string":
        query = f'''SELECT *
                    FROM tweets
                    WHERE content LIKE '%{keyword}%' AND timestamp BETWEEN '{start_date}' AND '{end_date}'
                    ORDER BY timestamp DESC;'''
    elif search_type == "hashtag":
        query = f'''SELECT t.* FROM tweets t
                    JOIN tweet_hashtags th ON t.tweet_id = th.tweet_id
                    JOIN hashtags h ON th.hashtag_id = h.hashtag_id
                    WHERE h.hashtag_text = '{keyword}' AND t.timestamp BETWEEN '{start_date}' AND '{end_date}'
                    ORDER BY t.timestamp DESC;'''
    elif search_type == "user":
        query = f"SELECT * FROM tweets WHERE user_id = {keyword} AND timestamp BETWEEN '{start_date}' AND '{end_date}' ORDER BY timestamp DESC;"
    cursor.execute(query)
    return cursor.fetchall()
```

PICTURE 3.2.1. A set of routine for searching applications

4.2 Performance Evaluation of Searches

4.2.1 Test of timings



```
def test_query_performance(conn,sqlstrings):
    test_queries = sqlstrings
    for query in test_queries:
        start_time = time.time()
        cursor = conn.cursor()
        cursor.execute(query)
        cursor.fetchall()
        end_time = time.time()
        print(f"Query:{query} took {end_time - start_time:.4f} seconds")
```

PICTURE 3.2.1. Query Performance Test Results

4.2.2 The Impact of Information Storage on Query Response Times

4.2.3 Trade-off of DuckDB

Because DuckDB's primary focus is on analytical processing (OLAP) rather than transactional processing (OLTP), it's powerful for complex searches and drill-downs but it may not be the best choice for applications requiring high concurrency and real-time data updates.