

STAT 694: Database Management for Advanced Data Science Applications

Final Project

Twitter Search Application - Team 31

[Github Repository Link](#)



RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

Team Members

Sasank Chindirala(sc2767)

Department of Statistics & Data Science

Rutgers University

New Brunswick, New Jersey

Supervised by

Prof. **Ajita John**

Abstract

This project involves the design and implementation of a data-driven application focused on efficiently managing and querying Twitter data. The system leverages the strengths of both relational and non-relational database technologies to manage user profiles and tweet data effectively. A comprehensive full-stack search application was developed to provide various search functionalities capable of handling large volumes of data. The architecture of the system was meticulously chosen to maximize scalability and processing speed, handling data in real-time as it streams from Twitter. This report provides an in-depth examination of the system architecture, data models, design choices, storage optimizations, and the performance of the implemented system.

1. Introduction

The primary objective of this project was to construct a robust backend system capable of storing and querying extensive Twitter datasets in a manner that mimics real-time data flow. This involved processing large amounts of structured and semi-structured data, requiring a system architecture that could handle high throughput and provide rapid access to data. To achieve this, PostgreSQL was employed for user profile management, while MongoDB was utilized for tweet data storage. The system utilizes FastAPI as the search application framework to enhance the application's responsiveness and support asynchronous operations, which are crucial for real-time data handling. Additionally, React was chosen as the frontend framework to provide a user interface reminiscent of Twitter, ensuring a user-friendly and intuitive search experience. The application allows users to perform searches based on usernames, hashtags, and specific tweet content, with informative drilldowns and options to further refine results through time-based filters. Moreover, a custom caching solution was implemented using Python to facilitate faster data retrieval. The following sections delve into the system's architecture, detailing the rationale behind each technology choice, and evaluate the impact on the system's performance through extensive testing of its search functionalities and caching capabilities.

2. Dataset

The dataset comprises a collection of tweets extracted from Twitter, stored as individual JSON objects in the files corona-out-2 and corona-out-3. Each json object encapsulates plenty of information about the user and their interactions within the platform. It contains information such as the creation timestamp, tweet ID, and user details, providing context about when and by whom the tweet was posted. The text field holds the actual content of the tweet and the entities field contains information regarding the hashtags referenced in the tweet. Alongside these primary details, supplementary metadata like the favorite count and retweet count enrich the dataset. A detailed view of the json data object is shown in Fig 1.

```
{
  "created_at": "Sun Apr 12 18:27:25 +0000 2020",
  "id": 1249403767180668930,
  "id_str": "1249403767180668930",
  "text": "RT @nuffsaidny: wishing death on people is weirdo behavior.",
  "source": "\u003ca href=\"http://twitter.com/download/iphone\" rel",
  "truncated": false,
  "in_reply_to_status_id": null,
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id": null,
  "in_reply_to_user_id_str": null,
  "in_reply_to_screen_name": null,
  "user": {
    "id": 1586716045552,
    "id_str": "1586716045552",
    "name": "Nuff Said NY",
    "screen_name": "nuffsaidny",
    "location": "New York City",
    "description": "I'm just a girl who loves to talk.",
    "url": null,
    "entities": {
      "url": {}
    },
    "protected": false,
    "followers_count": 1,
    "friends_count": 1,
    "listed_count": 0,
    "created_at": "Sun Apr 12 18:27:25 +0000 2020",
    "favourites_count": 0,
    "retweet_count": 0,
    "reply_count": 0,
    "quote_count": 0,
    "profile_image_url": "https://t.co/...",
    "profile_image_url_https": "https://t.co/...",
    "profile_banner_url": "https://t.co/...",
    "profile_link_color": "18771f",
    "profile_text_color": "000000",
    "profile_background_color": "f0f0f0",
    "profile_background_image_url": null,
    "profile_background_image_url_https": null,
    "profile_background_tile": false,
    "profile_use_background_image": true,
    "default_profile": false,
    "default_profile_image": false,
    "is_verified": false,
    "is_private": false,
    "is_blue_verified": false,
    "profile_interstitial_type": "none"
  },
  "geo": null,
  "coordinates": null,
  "place": null,
  "contributors": null,
  "retweeted_status": {
    "id": 1249315454797168641,
    "id_str": "1249315454797168641",
    "text": "wishing death on people is weirdo behavior.",
    "source": "\u003ca href=\"http://twitter.com/download/iphone\" rel",
    "truncated": false,
    "in_reply_to_status_id": null,
    "in_reply_to_status_id_str": null,
    "in_reply_to_user_id": null,
    "in_reply_to_user_id_str": null,
    "in_reply_to_screen_name": null,
    "user": {
      "id": 1586716045552,
      "id_str": "1586716045552",
      "name": "Nuff Said NY",
      "screen_name": "nuffsaidny",
      "location": "New York City",
      "description": "I'm just a girl who loves to talk.",
      "url": null,
      "entities": {
        "url": {}
      },
      "protected": false,
      "followers_count": 1,
      "friends_count": 1,
      "listed_count": 0,
      "created_at": "Sun Apr 12 18:27:25 +0000 2020",
      "favourites_count": 0,
      "retweet_count": 0,
      "reply_count": 0,
      "quote_count": 0,
      "profile_image_url": "https://t.co/...",
      "profile_image_url_https": "https://t.co/...",
      "profile_banner_url": "https://t.co/...",
      "profile_link_color": "18771f",
      "profile_text_color": "000000",
      "profile_background_color": "f0f0f0",
      "profile_background_image_url": null,
      "profile_background_image_url_https": null,
      "profile_background_tile": false,
      "profile_use_background_image": true,
      "default_profile": false,
      "default_profile_image": false,
      "is_verified": false,
      "is_private": false,
      "is_blue_verified": false,
      "profile_interstitial_type": "none"
    },
    "geo": null,
    "coordinates": null,
    "place": null,
    "contributors": null,
    "retweeted_status": null,
    "quoted_status": null,
    "quoted_status_id": null,
    "quoted_status_id_str": null,
    "quoted_status_permalink": null,
    "is_quote_status": false,
    "quote_count": 0,
    "reply_count": 0,
    "retweet_count": 0,
    "favorite_count": 0,
    "entities": {
      "hashtags": {},
      "urls": {},
      "mentions": {},
      "media": {}
    },
    "favorited": false,
    "retweeted": false,
    "filter_level": "low",
    "lang": "en",
    "timestamp_ms": "1586716045552"
  },
  "is_quote_status": true,
  "quote_count": 0,
  "reply_count": 0,
  "retweet_count": 0,
  "favorite_count": 0,
  "entities": {
    "hashtags": {},
    "urls": {},
    "mentions": {
      "screen_name": "nuffsaidny",
      "id": 1586716045552,
      "id_str": "1586716045552"
    },
    "media": {}
  },
  "favorited": false,
  "retweeted": false,
  "filter_level": "low",
  "lang": "en",
  "timestamp_ms": "1586716045552"
}
```

Fig 1. The JSON object received from the dataset.

Furthermore, the dataset incorporates features that distinguish retweets and quoted tweets. When a tweet is a retweet (identified by the text field starting with "RT"), the dataset includes information about the original tweet being shared, including its text, timestamp, user, and additional metadata. Similarly, in cases where a tweet quotes another tweet, the dataset includes details about the quoted tweet, including its text, timestamp, user, and metadata. Additionally, complete information about the user who tweeted the original content is provided, offering context for the quoted material. By incorporating information about retweets and quoted tweets, the dataset provides a comprehensive perspective on Twitter interactions, facilitating research into various aspects of information diffusion, user engagement patterns, and the dynamics of online discourse on the platform. A comprehensive summary, encapsulating the number of unique users, tweets and other pertinent high-level details extracted from the dataset can be found in Fig. 2 and Fig. 3 below.

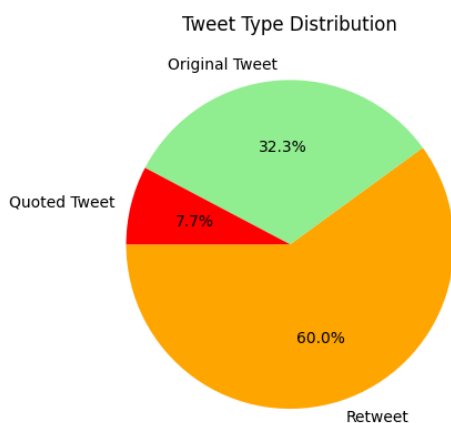


Fig. 2. Distribution of tweet types

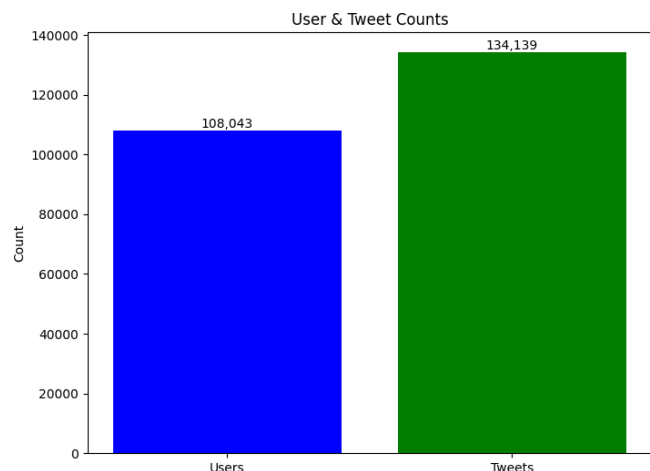


Fig. 3. Total unique users & tweets

3. Data Model & Datastores

3.1 User Data - PostgreSQL (Relational)

The user data from Twitter, such as user IDs, names, and profile-related metrics, is consistent and structured. PostgreSQL excels in handling such structured data due to its powerful relational data model. The attributes like 'followers_count', 'friends_count', and 'created_at' timestamp needed strict data integrity and consistency that Postgres could provide. Postgres is

highly extensible with the support for advanced indexing techniques like partial indexes, expression indexes, and the ability to use different index types, such as GIN (Generalized Inverted Index) and GiST (Generalized Structured Tree). These indexing mechanisms are ideal for full-text searching and indexing array data and this was one of the pivotal factors taken into consideration for the choice of the database. In addition, the strong support for ACID properties, advanced join capabilities and optimization for complex queries were among other reasons for choosing PostgreSQL to store user data.

3.1.1 User Data Model

The schema was chosen to include fields for user identification, profile details, tweets, followers and social graph metrics thereby dropping unnecessary attributes from the original user json object received from the dataset. Storing only the required attributes had clear practical advantages like reduced disk space requirement, lower memory usage, enhanced indexing efficiency and simplified data management. Moreover, an additional attribute called ‘tweets_count’, that was being used for relevance and ranking of username searches (discussed in detail in Section 5.1), was added directly into the user data model to avoid any on-the-fly computations. The model schema with all attributes is shown in Fig 4. below.

Column	Type
id	character varying(255)
name	character varying(255)
screen_name	character varying(255)
verified	boolean
location	character varying(255)
description	character varying(255)
followers_count	bigint
friends_count	bigint
favourites_count	bigint
statuses_count	bigint
tweets_count	bigint
created_at	timestamp without time zone

Fig. 4. Schema of user data stored in PostgreSQL

3.1.2. User Data Storage

The data processing and storage was achieved using a 'user-data-processor' Jupyter notebook, which was equipped with all necessary libraries, including `psycopg2` and `json`, among others. This notebook facilitates the establishment of a connection to the PostgreSQL engine, enabling the creation of a 'twitter' database and a 'users' table according to the predefined schema, as well as the addition of user records to this table. Data ingestion is managed through the 'load_user_data_to_database()' function, which processes the corona-out-2 and corona-out-3 files. This function systematically reads each file line-by-line to parse individual JSON objects. Upon successfully reading each object, it extracts user details and calls the 'load_unique_user()' function to ascertain whether the user already exists within the database. If the user is not previously stored, the 'insert_user()' function is triggered to add the new user to the database. A key observation here is that the JSON objects include not only direct user information but also nested fields - specifically 'retweet_status' and 'quoted_status' - that potentially contain additional user data. When these nested fields are present and include user information, they are processed in a manner similar to the direct user data, ensuring comprehensive data capture in the database.

After successfully ingesting the user data into the database, indexing was added to improve the search functionality efficiency. Considering the nature of the searches performed (typically wildcard searches, for. e.g. where screen_name like '%keyword%'), an inverted index was added to the 'screen_name' column. Another multi-column index was added on columns 'followers_count', 'tweets_count' and 'verified' that was being used for sorting the search results (relevance). The indexes defined on the 'users' table can be seen in Fig 5.

Indexes:

```
"users_new_pkey" PRIMARY KEY, btree (id)
"idx_users_ranking" btree (followers_count DESC, tweets_count DESC, verified DESC)
"idx_users_screen_name" gin (screen_name gin_trgm_ops)
```

Fig. 5. Indexes defined on the user table

3.2 Tweet Data - MongoDB (Non-Relational)

Tweets are highly dynamic and can contain various types of content including text, images, videos, URLs, and a varying array of hashtags. Each tweet is essentially a semi-structured document. MongoDB's schema-less nature allows for this kind of flexibility, accommodating any changes in data structure without requiring schema migrations. It stores data in BSON (Binary JSON), which is highly suitable for the JSON-like structure of tweets. Moreover, it is designed to scale horizontally through sharding, effectively handling large datasets and high write loads, which is typical for tweet data. Finally, given that tweets involve a lot of text data and the need for text search (by hashtag, content, etc.), MongoDB's full-text search capabilities and provided built-in text search functionality, was crucial for efficiently implementing features like searching tweets based on content or hashtags making this an obvious choice for storing the tweet data.

3.2.1 Tweet Data Model

The tweet data model design utilizes a document-oriented approach, which aligns with the inherent structure of tweets. The document stores attributes of each tweet, including the tweet ID, text, hashtags, creation timestamp and other relevant information. Additionally, each tweet is mapped to the corresponding user by storing the user ID and user name for that tweet within the 'tweets' collection. In the case of retweets or quoted tweets, the ID of the source tweet is stored in the document, facilitating easy mapping between retweets or quoted tweets and their originals. Furthermore, if a tweet is identified as a retweet, the collection also increments the retweet count for the source tweet document. This ensures that the original tweet's retweet count is updated correctly. Another field, named 'tweet_score', is added to each document object to take a weighted average of the fields 'likes_count' and 'retweet_count'. This enhances the relevance and ranking of searches utilizing this collection and safely avoids any runtime computations needed for this purpose. The document design with all the fields is shown in Fig 6. below.

```

_id: ObjectId('662964076cc8b70cf9abe575')
tweet_id: "1249402922309423107"
text: "In Turkey, there are 300 thousand prisoners and 150 thousand prison em..."
▶ hashtag: Array (empty)
user_id: "1055885344736993280"
user_name: "no Comment"
user_screen_name: "lastcavalry61"
likes_count: 5
retweet_count: 21
source_tweet_id: 0
tweet_score: 14.6
created_at: "2020-04-12 18:24:04"

```

Fig. 6. Document design of tweet data stored in MongoDB

3.2.2 Tweet Data Storage

The processing and storage of tweet data are executed through a Jupyter notebook titled 'tweet-data-processor,' which is loaded with all necessary libraries, including pymongo and json. This notebook contains the code necessary to connect to the MongoDB engine, facilitating the creation of the 'twitter' database and the 'tweets' collection. This setup stores documents and fields aligned with the tweet data model and supports the loading of tweets into this collection. Tweet data is loaded into the collection via the 'load_tweet_data_to_database()' function, which reads data from dataset files line by line and performs ingestion. This process involves the execution of several functions; the 'tweet_exists()' function which verifies the uniqueness of tweets in the collection to avoid duplication, the 'update_tweet()' function which increments the retweet count for existing tweets when the 'text' field of the current tweet starts with 'RT' and the source tweet is already present in the collection, the 'insert_tweet()' function which finally adds the tweet as a document to the collection.

Upon successful completion of loading the tweets into the database, a couple of indexes were added to optimize the supported searches (discussed in Section 5). A text index was added on the 'text' field to improve performance when searching for specific words or phrases within string content. A compound index was added on the 'user_screen_name' and 'tweet_score' fields to optimize the drill down feature of fetching user specific tweets. The indexes defined on the 'tweets' collection can be viewed in Fig 7.

Name and Definition	Type	Size	Usage	Properties
> _id_	REGULAR ⓘ	2.7 MB	8 (since Wed Apr 24 2024)	UNIQUE ⓘ
> text_text	TEXT ⓘ	28.8 MB	29 (since Wed Apr 24 2024)	
> user_screen_name_1_tweet_score_-1	REGULAR ⓘ	2.6 MB	12 (since Sun Apr 28 2024)	COMPOUND ⓘ

Fig. 7. Indexes on the tweets collection

4. Cache

A caching solution was implemented using a python dictionary to store tweets and user data that is retrieved from the searches. A cache class was created and was equipped with features that are expected from caching systems in handling real world challenges like long retention of stale data, performance degradations due to over utilization and data losses from system crashes. The methods implemented aim to address each of the above mentioned challenges while enriching the search experience with efficient sub-millisecond latencies.

4.1 Initialization and Configuration

The ‘`__init__()`’ method handles all the required initializations for the Cache class. A cache object for this class can be initialized using the following required parameters;

‘`max_size`’: Maximum number of items the cache can hold.

‘`evict_strategy`’: Strategy for evicting items from the cache when it's full.

‘`checkpoint_interval`’: Interval in seconds for saving the cache state to disk.

‘`ttl`’: A time-to-live for cache items in seconds, after which items are considered expired and removed.

Considering the scale and the sizes of the ‘users’ and ‘tweets’ databases, a default set of configurations were chosen; ‘`max_size`’ to store 10000 items which roughly could account to around 5MB of data, ‘`evict_strategy`’ to flush the least recently used item when the cache reaches its maximum capacity, ‘`checkpoint_interval`’ of 10 minutes (600 seconds), after which the latest state of the cache is stored to disk, and ‘`ttl`’ set to None (for simplicity), implying the cached data doesn’t flush values until the cache reaches its maximum capacity or an entry is flushed manually.

4.2 Persistence

A couple of methods are implemented to persist the state of the cache onto disk and to retrieve the latest state of the cache from the disk. The 'load_from_checkpoint()' method loads the cache and its access count from a file. The 'save_to_checkpoint()' method saves the cache state and access counts to a file at regular intervals defined by checkpoint_interval. The pickle module was used for the serialization of the dictionary object to the file and the deserialization of the data from the file back into a dictionary object.

4.3 Cache Operations

Changes to the cache are done through the methods defined here. New data is either stored into the cache using the 'put()' method and cached data is retrieved using the 'get()' method. If the 'ttl' is set to a value other than None during initialization, then the 'expire_items()' method is used to remove items from the cache that have exceeded their TTL.

5. Search Application Design

In a nutshell, the high level system architecture can be visualized as shown in Fig. 8 below.

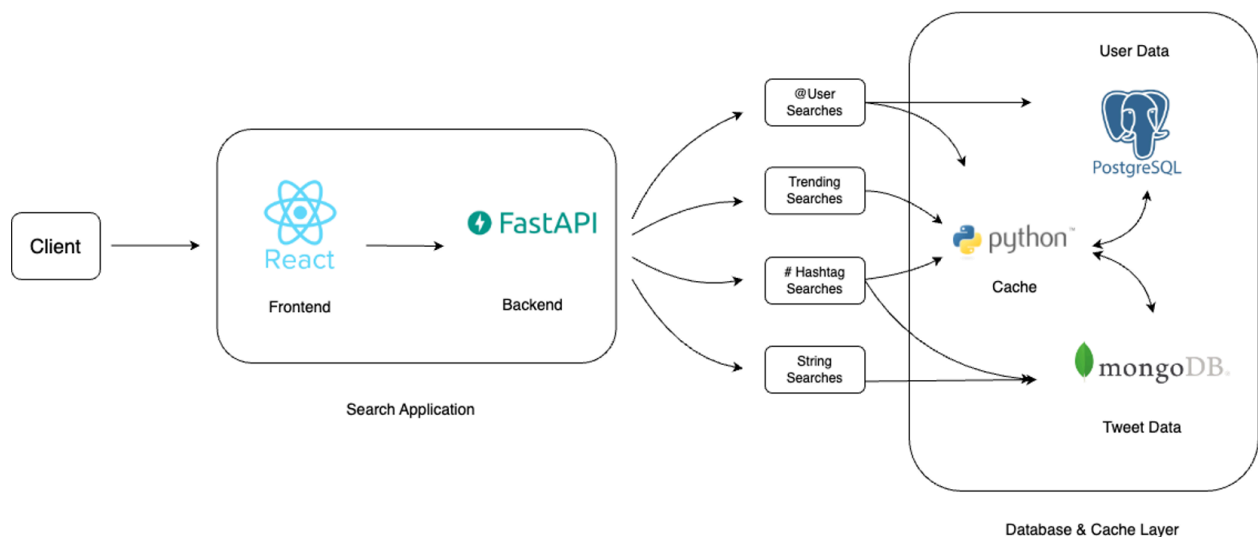


Fig. 8. Search Application Architecture

The different types of searches supported by the system are detailed in the following subsections. As discussed in sections 3.1.1 (User Data Model) and 4.1.1 (Tweet Data Model), the relevance of the searches was handled through additional attributes or fields stored directly in the respective data models, reducing any added complexity during data retrieval. Moreover, the indexes on the 'users' table and the 'tweets' collection reduce any latencies involved in the search process.

5.1 User Search

Searching for users is fundamental to our application. When initiating a user search, users prefix their query with an "@" symbol, signaling a search for user profiles. The application's search function, optimized for performance, first checks the cache for any matching results. If found, these results are immediately retrieved. If not, the query is processed against the PostgreSQL database, and results are cached for subsequent searches. The search results include user information with a drill down feature fetching the most recent tweets for each user, fetched from MongoDB, enhancing the search experience by providing a snapshot of their recent activity.

5.2 Hashtag Search

Hashtag searches start with a "#" symbol followed by the desired hashtag. This query taps into MongoDB's robust text search capabilities to retrieve and display tweets that reference the hashtags that match the query, ranked by the 'tweet_score' relevance metric defined on the tweet data. This process is handled efficiently in the background, pulling data from either the cache or directly from MongoDB, ensuring a seamless user experience by providing instant results.

5.3 String Search

String search is vital for discovering content within tweets. The application uses a text index on the tweet's text field to enhance query efficiency. Queries exclude common stopwords to refine relevance, focusing on substantial keywords. If a search query does not begin with special characters like '#' or '@', it defaults to a string search. This search type is particularly dynamic, with results sorted by the 'tweet_score' metric. Given the high variability and specificity of string searches, results are not cached, ensuring that only the most current data is presented.

5.4 Trending Search

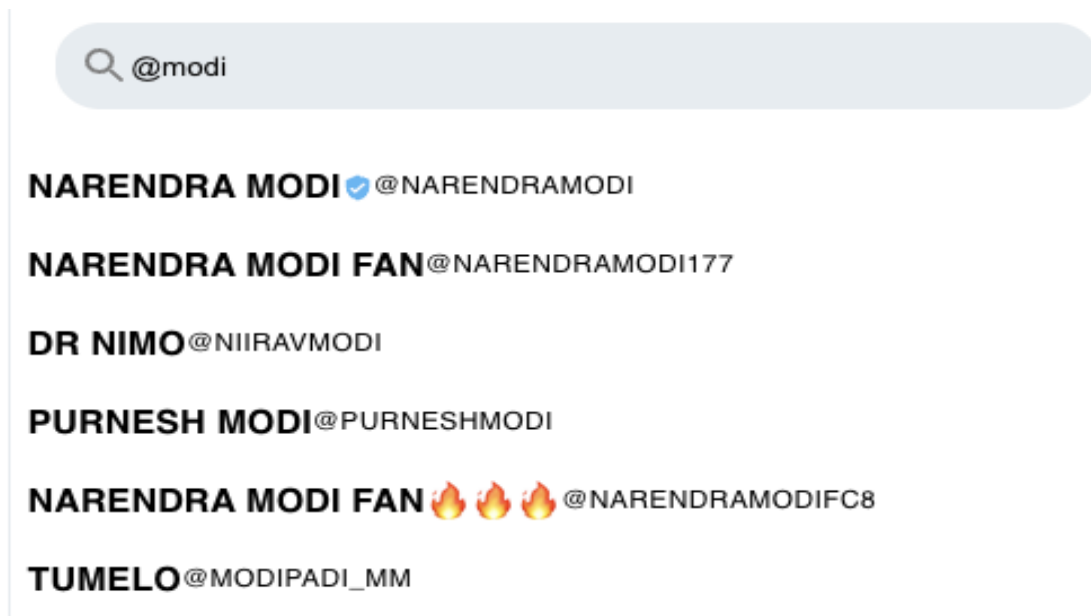
Trending searches could encompass all three types of searches mentioned above. Hence, the

application was designed to provide quick views into all the three types of trending data; trending hashtags, trending users and trending tweets. The trending hashtags results are loaded directly onto the home page of the application and the trending users and trending tweets can be navigated directly from the home page. All types of trending data is cached for an application wide performance boost.

These search functionalities collectively enhance the usability of the application, making it a powerful tool for navigating and extracting value from the vast streams of Twitter data. Each search type is tailored to specific user needs, from exploring user profiles to diving into the nuances of tweet content and discussions.

6. Results

6.1 User Search




Search Latency Without Cache





0.04187798500061035 seconds





Search Latency With Cache


2.7179718017578125e-05 seconds

6.2 Hashtag Search

 #corona

**Frank Figliuzzi** @FrankFigliuzzi1 13 Apr, 2020 02:10PM
Mob boss asks his crew if that guy who likes science is a snitch.
#fauci #corona
 1  9217 

**Chad Ellsworth** @chad_ellsworth 19 Apr, 2020 06:53PM
Happening Now in San Diego in response to closing walking trails and beaches. #COVID19 #corona #encinitas #sandiego
<https://t.co/ukmyMbx2yq>
 1  8097 

**Abdullah T.R #coronavirus** @TraderAT12 22 Feb, 2020 04:09PM
#coronavirus #corona


Search Latency Without Cache





0.07761096954345703 seconds





Search Latency With Cache

2.7894973754882812e-05 seconds

6.3 String Search

 doctors

**Pawan Kalyan** @PawanKalyan 22 Mar, 2020 11:40AM
We salute to all the Doctors, Nurses, health workers, sanitary workers, media and police for fighting against coron... <https://t.co/gb6eQWgINp>
 1  76808 

**Chiranjeevi Konidela** @KChiruTweets 20 Apr, 2020 04:11PM
We can never thank them enough, all our doctors, health workers, police, sanitation workers and media, our frontli...
<https://t.co/HR4KAeh76s>
 1  11289 

Search Latency Without Cache

0.030508995056152344 seconds

Search Latency With Cache

4.506111145019531e-05 seconds

6.4 Trending Search

6.4.1 Trending Hashtags

Search Latency Without Cache

0.18890810012817383 seconds

Search Latency With Cache

1.5974044799804688e-05 seconds

What's happening

Trending hashtags

#Corona 5.92K

#corona 1.93K

#Mattarella 1.52K

#25Aprile 1.48K

#Covid_19 1.12K

6.4.2 Trending Tweets

Search Latency Without Cache

0.11109113693237305 seconds

Search Latency With Cache

1.621246337890625e-05 seconds



Pre K @stayfrea_ 04 Mar, 2020 05:31PM

ALERT!!!!!!

The corona virus can be spread through money. If you have any money at home, put on some gloves, put al... <https://t.co/fuJjDpFN3I>



1



1128502



gilbert @HtownBabyG 13 Mar, 2020 12:43AM

corona virus enters my body

The 4 Flintstone gummies I ate in 2005: <https://t.co/3STfdlQtaT>



1



811062



6.4.3 Trending Users

Search Latency Without Cache

0.0077250003814697266 seconds

Search Latency With Cache

4.673004150390625e-05 seconds

BARACK OBAMA @BARACKOBAMA

DONALD J. TRUMP @REALDONALDTRUMP

CNN BREAKING NEWS @CNNBRK

NARENDRA MODI @NARENDRAMODI

SHAKIRA @SHAKIRA

CNN @CNN

THE NEW YORK TIMES @NYTIMES

BBC BREAKING NEWS @BBCBREAKING

AMITABH BACHCHAN @SRBACHCHAN

SALMAN KHAN @BEINGSALMANKHAN

7. Conclusions

This project provided a valuable opportunity to examine data integration across various technologies. It involved importing data from a JSON file into two distinct database systems, PostgreSQL and MongoDB. Additionally, a FastAPI-based search application with a frontend using React was developed to facilitate data retrieval from these databases based on specific user queries. The project also emphasized the importance of leveraging caching mechanisms through Python dictionaries and advanced database indexing to enhance query efficiency. By implementing search result caching and table indexing, the response time for queries was significantly reduced, leading to a more streamlined and effective system. Overall, this project served as an excellent learning experience in combining different technologies and underscored the critical role of database architecture and optimization in boosting the performance of database systems.

8. References

1. <https://www.mongodb.com/docs/manual/core/indexes/index-types/index-text/>
2. <https://www.mongodb.com/docs/manual/core/indexes/index-types/index-compound/>
3. <https://www.postgresql.org/docs/current/indexes-types.html>
4. <https://www.postgresql.org/docs/current/indexes-multicolumn.html>
5. <https://fastapi.tiangolo.com/tutorial/>
6. <https://medium.com/@saleem.latif.ec/implementing-a-custom-cache-in-python-68c39ece8a8>