**TEAM 4**

**Covid-19 Twitter Search API**

**MSDS Data Science 16: 954:694:01**

**Instructor: Dr. Ajita John (ajita.john@rutgers.edu)**

**TA: Zhe Zhang (zz323@stat.rutgers.edu)**

**Yunjiao Bai (yb241@stat.rutgers.edu)**

**Chandra Bhushan Gupta – cg886**

**Shubham Kokane – ssk203**

**Rutvik Deshpande – rvd39**

**Priyal Shaha – ps1274**

[Team\_4\_Codebase](https://github.com/shubhamkokane/694_2023_Team4)

**Introduction**

Twitter is one of the key social media platforms with millions of active daily users engaging via Tweets and Retweets. With such a vast user base, the Twitter API provides ample opportunities for data exploration and analysis.

The project aims to organize Twitter data by building ingestion mechanisms into both NoSQL and SQL databases. Twitter data was provided and ingestion layers were constructed with the assumption that real-time tweet streams would be ingested. Search APIs were exposed using the Python Flask API with dictionary-based caching to speed up various search combinations.

Overall, the project provided an opportunity for the team to gain hands-on experience with the Twitter API, Python programming, and database management with Cassandra and MySQL. It also demonstrated the importance of data storage and retrieval in real-world applications.

**Assumptions and Dependencies**

The team utilized a dataset provided by professor and processed it through the ingestion layer, assuming a real-time streaming process tweet by tweet. Various dependencies, including Docker containers for Cassandra, Kafka, and MySQL, and containerized Flask API were leveraged to build the architecture.

**Dataset**

|  |  |
| --- | --- |
| created\_at | Timestamp when the tweet was created |
| id | Unique identifier for the tweet |
| text | Content of the tweet |
| user | A collection of user objects that represent the user who posted the tweet |
| quote\_count | Number of times the tweet has been quoted |
| reply\_count | Number of times the tweet has been replied to |
| retweet\_count | Number of times the tweet has been retweeted |
| favorite\_count | Number of times the tweet has been favorited |
| retweeted | Boolean value indicating whether the authenticated user has retweeted this tweet |
| lang | Language of the tweet |
| timestamp\_ms | Timestamp when the tweet was created in milliseconds |
| user\_id | Unique identifier for the user |
| name | User's display name |
| screen\_name | User’s screen name |

The Twitter API provides access to various types of data, including tweets, user profiles, retweets and multiple other metrices. Tweets contain various attributes such as tweet text, user information, hashtags, and timestamps. User profiles include information such as the user’s bio, location, followers, and tweets.

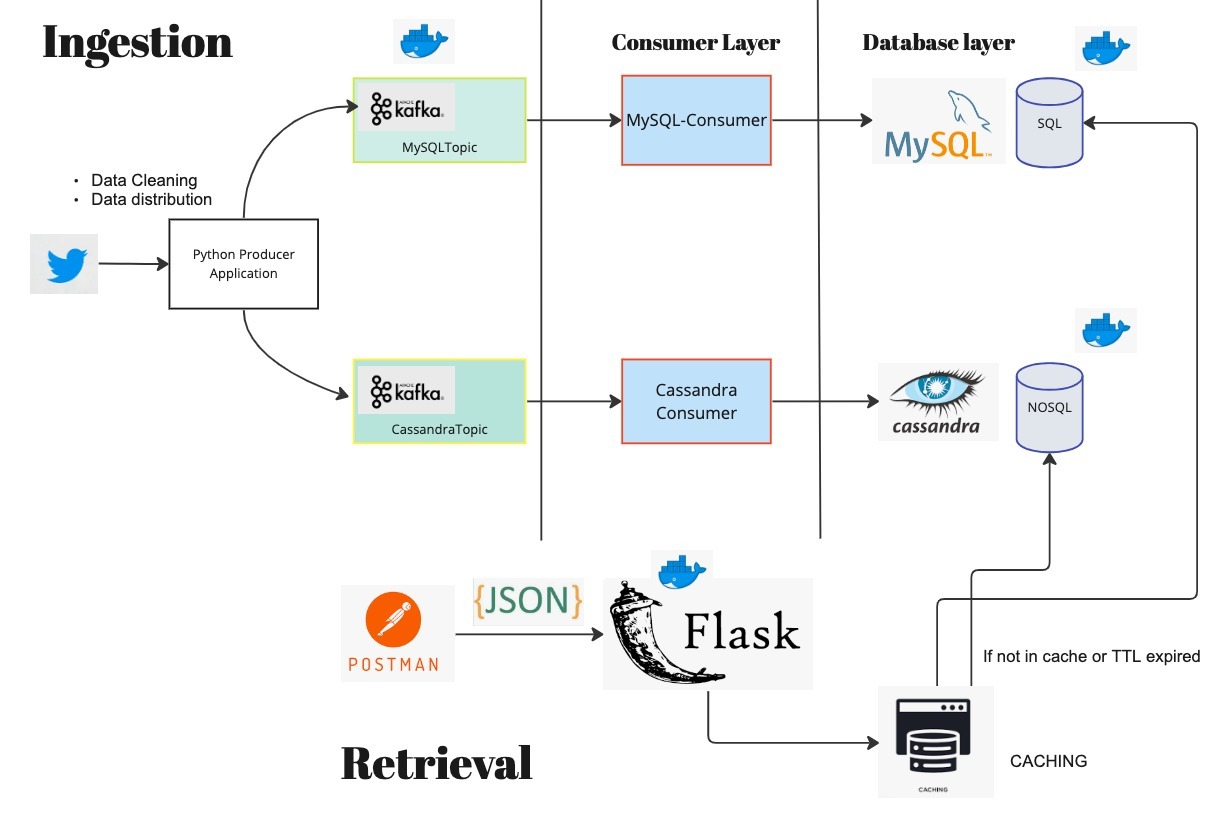
|  |  |
| --- | --- |
| **Original Tweets** | **51K** |
| **Unique #Hashtags** | **6.7K** |
| **Unique Users** | **81K** |
| **Retweets** | **61K** |

To clean the data, several preprocessing steps were performed, depending on the specific research question or analysis goals. Action was performed by removing duplicates, removing irrelevant or incomplete data, correcting errors, standardizing the data format, and performed NLTK search on tweet\_text to identify the hashtags in the tweet response that weren’t captured in the hashtags JSON object. Some of the key observations:

The tweet object data contains over 20 unique attributes. For some specific use-cases, the following fields were derived: id, tweet\_text, user\_id, quote\_count, reply\_count, retweet\_count, favorite\_count, created\_at, and language.

**Architecture**

The project followed an event-driven architecture to ensure the system remains highly available, fault-tolerant, and performant when handling a continuous stream of tweets. The system is composed of several components, each with a specific function. To achieve parallel processing, SQL and Cassandra consumers were separated into their respective topics.

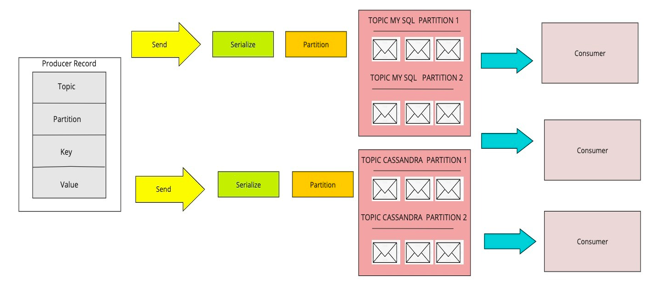


|  |  |
| --- | --- |
| **Docker** | The use of Docker containers allows for easy deployment and scaling of the application. |
| **Python Kafka producer** | Python Kafka producers to publish tweets, which were then consumed |
| **MySQL consumer** | The SQL consumer was responsible for storing the user information and the tweet information into a MySQL database |
| **Cassandra consumer** | Cassandra consumer was responsible for storing the tweets into a Cassandra NoSQL database. |
| **My SQL DB** | Relational component to store normalized data with joins, indices, and other DB objects |
| **Cassandra DB** | NoSQL component which is highly available and fault tolerant |
| **Flask API** | FLASK API was used to facilitate REST based API for search functionality and support HTTP requests |
| **JSON** | JSON format was used to send data in Kafka topic as well as response from Flask API. |
| **POSTMAN** | Postman, a popular tool for testing APIs, was used to test and verify the system's functionality. |
| **CACHING** | Python dictionary-based caching was implemented to improve the search functionality |

**Docker**

Docker is used to help team members to package applications and dependencies into containers. Docker compose is used to build Docker containers for Cassandra, Kafka, and MySQL to simplify deployment and ensure consistency across different environments.

**Kafka**

SQL topic is used to create a table in a relational database, such as MySQL, to replicate the structure of tweet data. Whenever a new tweet was published to the "tweets" Kafka topic, it was written to the corresponding table in the database.

Tweet data was stored in a NoSQL database, Apache Cassandra using a Cassandra Topic. When a new tweet is published to the "tweets" Kafka topic, it is written to the Cassandra Topic as a new row. The schema of the Cassandra Topic was designed to optimize query performance for the specific use case, such as retrieving tweets based on their hashtag or tweet\_text. In both cases, Kafka acted as a message broker between the Twitter API JSON response and the database, enabling the system to scale horizontally and process large volumes of data in real-time.

|  |  |
| --- | --- |
| Kafka Topic creation |  |
| Kafka producer code that ran inside a loop to continuously send tweets as a stream |  |
| Kafka consumer  code for data ingestion in cassandra | A screen shot of a computer  Description automatically generated with low confidence |
| Kafka consumer  code for data ingestion in MySQL |  |

**MySQL Data Model**

Diagram

Description automatically generatedThe data model for the Twitter JSON response stored in a MySQL datastore includes two tables: a "user" table and a "user\_count" table. The "user" table contains attributes such as the user\_id, name, screen name, location, and created\_at. The "user\_count" table contains aggregate statistics for each user, such as follower\_count, friend\_count, listed\_count, favorite\_count. The two tables are linked by the user\_id, which served as a foreign key in the "user\_count" table. With this data model, it is possible to perform various analytical queries to gain insights into user behavior on Twitter.

***User Data Model:*** The key attributes stored in the “user” table: user\_id, name, screen\_name, location, user\_created\_at, verified.

***User\_count Data Model:*** The key attributes stored in the “user\_count” table: user\_id, followers\_count, friends\_count, listed\_count, favorites\_count, statuses\_count.

**Cassandra Data Model**

Table

Description automatically generatedCassandra datastore contains three tables (tweets, retweets, and hashtags), the data model is designed to optimize query performance for the specific use case of retrieving tweets based on their hashtag and user\_id. The tweets table contains the tweet data, with the tweet\_id as the partition key and the user\_id as the clustering key. The retweets table contains retweet data, with the retweet\_id as the partition key and the retweet\_created\_at as the clustering key. The hashtags table contains data about the hashtags used in tweets, with the hashtag text as the partition key and the tweet\_id as the clustering key.

By using multiple partition and clustering keys, queries were optimized to efficiently retrieve tweets based on different criteria, such as tweet\_id and hashtag text. With the use of Cassandra, the data is partitioned across multiple nodes for high availability and scalability.

**A picture containing text, font, guide

Description automatically generatedCaching**

The caching mechanism implemented using Python dictionary utilized both Least Recently Used (LRU) and Time-To-Live (TTL) strategies to manage cache memory efficiently. When the cache limit was reached, the least recently used items were removed to make space for new items. Additionally, a Time-To-Live value was set for each cached item, after which the item was considered invalid and would be removed from the cache on the next access attempt. This combination of LRU and TTL strategies ensured that the cache only contained the most frequently and recently accessed items, while also preventing stale data from being retained indefinitely.

**Caching Strategy**

A picture containing text, font, screenshot, algebra

Description automatically generatedA Python dictionary-based caching mechanism with Least Recently Used (LRU) and Time-to-Live (TTL) features are implemented. The system used a dictionary to cache database calls' results, which allowed for quick access and reduced the number of redundant database calls.

To implement the TTL feature, each cache entry is given an expiration time. Whenever a key is accessed, the system checked its expiration time and removed it from the dictionary if it had expired. This ensured that the cache only contained valid entries.

Both the LRU and TTL features are used together to provide an efficient and self-maintaining caching mechanism. This implementation allowed for significant performance gains in applications that required frequent function calls with the same input parameters.

**Search API**

Search APIs were built using Flask, a popular Python web framework that allows developers to build web applications quickly and easily. It provides a simple and flexible way to create web applications and APIs using Python. Flask used HTTP to retrieve data from the server, and it is commonly used to implement search functionality in web applications.

Here are the high-level steps which were performed to build Search API with databases:

***Parsing the request:*** When Flask receives an HTTP request, it parses the request to extract the relevant information, such as the endpoint being accessed and any query parameters.

***Processing the request:*** Based on the endpoint and query parameters, the Flask API determines which data should be retrieved from the database. For example, if the endpoint is `/users` and the query parameter is `id=123`, the Flask API will retrieve the user with ID 123 from the database.

***Executing the query:*** Once the query has been identified, the Flask API executes it against the database to retrieve the requested data.

Returning the response: Once the data has been retrieved from the database, the Flask API formats it into an HTTP response and returns it to the client.

Output of the API was presented in formatted JSON on postman UI.

|  |  |
| --- | --- |
| Implemented to identify all the users in the dataset.  **Query:** *SELECT \* FROM users* |  |
| Endpoint implemented to find a specific user.  **Query:** *SELECT \* FROM users where name=%s* |  |
| Endpoint implemented to drill down specific users from a particular location\_id  **Query:** *SELECT \* FROM users where name=%s and location=%s* |  |
| Endpoint implemented to demonstrate the autocomplete feature to identify usernames starting with the given string.  **Query:** *SELECT \* FROM users where name LIKE %s* |  |
| Endpoint implemented to demonstrate the JOIN feature between two tables in the SQL datastore.  **Query:** *SELECT uc.\*, u.name FROM users u, user\_count uc where u.user\_id = uc.user\_id and u.name = %s* |  |
| Endpoint implemented to demonstrate the ORDER BY feature to demonstrate the top 10 users based on the follower\_count.  **Query:** *SELECT u.name, uc.followers\_count FROM users u, user\_count uc where u.user\_id = uc.user\_id ORDER BY followers\_count DESC LIMIT 10* |  |
| Endpoint implemented to demonstrate timeseries data using hashtags in NoSQL.  **Query:** *select \* from hashtags WHERE hashtag\_text = %s and hashtag\_created\_at > %s and hashtag\_created\_at < %s* |  |
| Endpoint implemented to find the tweets containing specific words and display those tweets.  **Query:** *SELECT tweet\_id, tweet\_text FROM tweets* |  |

|  |  |
| --- | --- |
| A screenshot of a computer  Description automatically generated with low confidence  A screenshot of a computer  Description automatically generated  Endpoint: [/hashtag?hashtag=coronavirus&start=2020-04-01&end=2020-04-30](http://127.0.0.1:5000/hashtag?hashtag=coronavirus&start=2020-04-01&end=2020-04-30)  Without Cache: 0.0139 seconds  With Cache: 3.099 e-06 seconds | A picture containing text, screenshot, black  Description automatically generated  A screenshot of a computer  Description automatically generated with medium confidence  Endpoint:  [/wordfind/help](http://127.0.0.1:5000/wordfind/help)  Without Cache: 0.846 seconds  With Cache: 2.598 e-05 seconds |
| A screenshot of a phone  Description automatically generated with low confidence  A screenshot of a computer  Description automatically generated with medium confidence  Endpoint: /user/shubham  Without Cache: 0.0641seconds  With Cache: 2.503 e-05 seconds | A screenshot of a computer  Description automatically generated with low confidenceA screenshot of a computer program  Description automatically generated with low confidence  Endpoint: /user/all  Without Cache: 1.1966 seconds  With Cache: 0.0003 seconds |

**Optimization**

***NoSQL:*** Queries were built around Partition Keys, Clustering Keys as per the use cases.

***SQL:*** Indexing on followers\_count, screenname; Regular monitoring of query performance was conducted using “Explain Plan” to identify the optimal indexing strategy for various columns in the MySQL datastore.

**Learnings**

***Data Ingestion:*** Learned how to consume and process data from an external data source like Twitter and store it in different data stores.

***Data Modeling:*** Learned how to create and maintain data models for SQL and Cassandra.

Data Processing: Learned how to process data in real-time using Kafka, including topics, brokers, and partitions.

***Caching:*** Learned how to implement an LRU (Least Recently Used) caching mechanism to improve application performance by keeping frequently accessed data in memory.

***Integration:*** Learned how to integrate different technologies like Flask, MySQL, Cassandra, Kafka, and caching in a single project.

Troubleshooting: Learned how to troubleshoot common issues that arise when working with different technologies and how to debug them.

In conclusion, this project provided valuable experience in working with a variety of technologies commonly used in data processing and web development. By combining these technologies, the team was able to build a scalable and efficient search functionality for users and tweets.

**Division of Work**

|  |  |
| --- | --- |
| **Chandra Bhushan Gupta**  Docker Configurations and architecture Search API with Flask  Code Organization  NoSQL Database modeling | **Shubham Kokane**  Kafka configuration  Kafka producer  Cassandra Consumer  Data cleaning and preprocessing |
| **Rutvik Deshpande**  SQL Database modeling  Postman setup  MySQL Kafka Consumer  NoSQL Database Setup | **Priyal Shaha**  Caching  Search API using Caching  Data cleaning and preprocessing  Kafka Topic Creation |

**References**

***https://kafka-python.readthedocs.io/en/master/***

***https://kafka.apache.org/***

***https://www.youtube.com/@Confluent***

***https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf***

***https://cassandra.apache.org/\_/index.html***