Team: BUG

> Tianhao Liu (20205784)
>
> Haotian Shi. (20205757)
>
> XIaoyu han (20205758)

Synopsis

Imagine us as the world's leading movie review platform, where users from all corners of the globe can express their opinions on various films. Our primary objective is to empower users to evaluate movies and generate personalized recommendation lists based on their preferences and statistical insights.

Due to the growing number of users, it's imperative to construct a distributed system to bolster the performance, scalability, and reliability of the system

Technology Stack

- **Kafka + Zookeeper Cluster**

  We utilize Kafka with Zookeeper for our message queue. This state-of-the-art distributed message queue technique enables us to effectively manage concurrent operations. By leveraging topics, we can differentiate between different messages, ensuring isolation between all microservices. Isolation is crucial, especially since we have adopted a microservice architecture, and different services may be implemented using different languages (for instance, we will deploy a machine learning model in one of our services, and Python is much more suitable than Java for this task). Additionally, the Kafka cluster is well-known for its scalability, which can be advantageous for future upgrades.

- **Redis**

Our project is data-driven, focusing on real-time movie reviews, which involves managing streaming data. With the substantial data queries required for calculating real-time movie ratings, Redis cache stands out as the optimal solution for this scenario. Additionally, Redis supports cluster mode, enabling horizontal scalability.

- **MongoDB**

  Our project also requires data backup for reviews and various types of data. MongoDB's document-oriented data model allows for flexible and dynamic schemas, making it very easy to use. Additionally, MongoDB is designed to scale out horizontally, meaning it can handle large volumes of reviews from people all over the world.

- **REST API**

  Since we have developed some frontend tools for visualization, utilizing a standardized REST API can effectively separate backend and frontend tasks, saving a significant amount of time in communication.

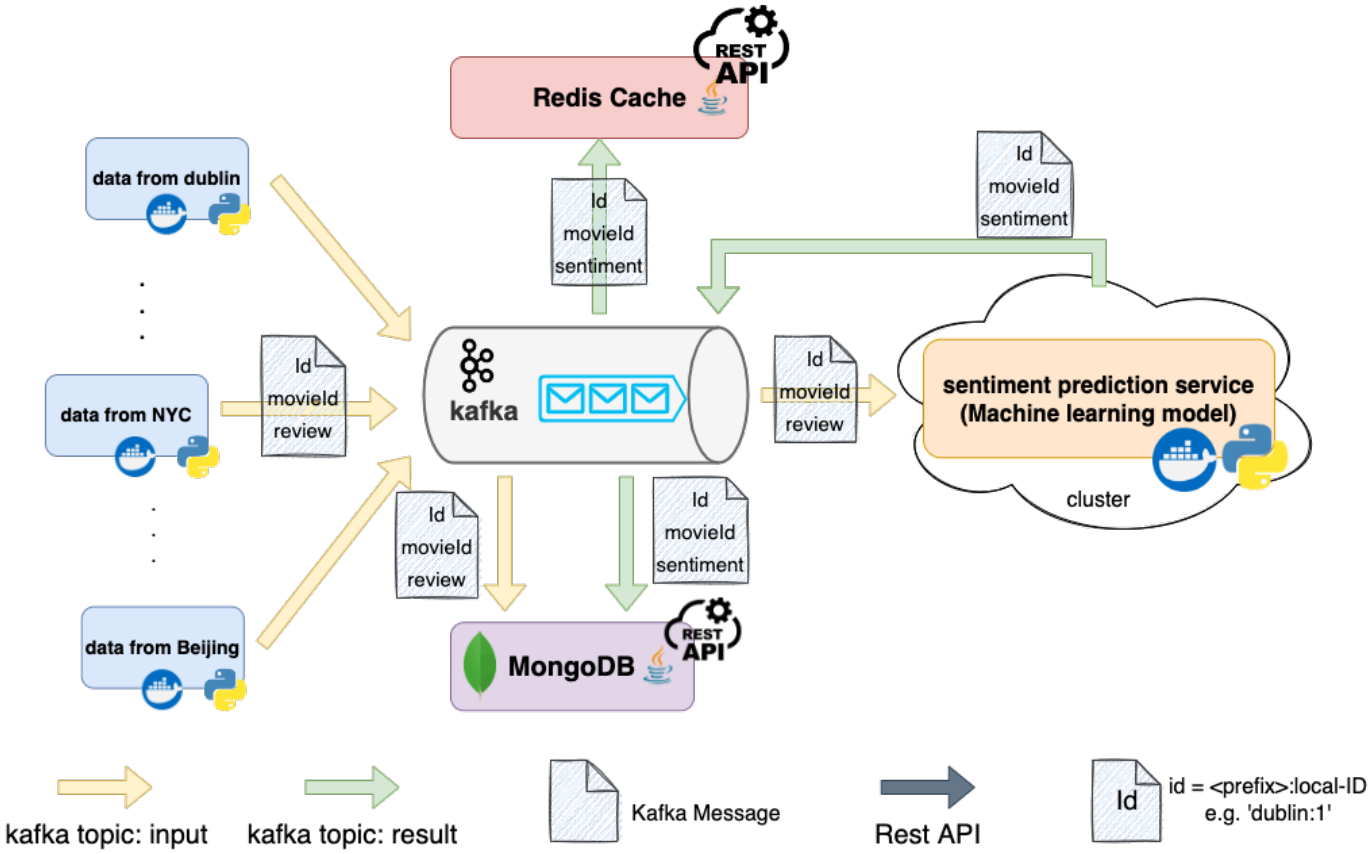
System Overview

Built upon the cutting-edge distributed system technologies widely employed in the industry, our project integrates a sophisticated tech stack to ensure seamless operation and exceptional performance. We utilize Kafka as our messaging queue, facilitating efficient communication and data flow between different components of the system.

To enhance responsiveness and speed, we employ Redis as a high-performance caching solution, optimizing data retrieval and delivery. MongoDB serves as our NoSQL storage, offering scalability and flexibility to accommodate the vast amounts of user-generated content.

Moreover, we embrace a modern approach to software architecture by implementing a separation between the front-end and back-end through REST APIs. This enables us to achieve greater modularity, scalability, and maintainability, fostering a robust and adaptable system capable of meeting

the evolving needs of our users.



Functionally, our system consists of four services. Here is a detailed description of each service and how they interact within the system.

Data Collection Service

- **Objective**

  Since we lack access to a real movie review website and find third-party APIs too costly for streaming data, we have opted to simulate real-time user reviews by continuously sending reviews from a database at random intervals. The data used can be sourced from the Kaggle 'IMDB Dataset of 50K Movie Reviews'.

- **Implementation**

This service will encapsulate each review into a JSON message, including a Global Review ID, movie ID, and review content. As the data source is distributed and multiple instances may be deployed across different countries, independence is essential. Using a simple anonymous increasing ID may lead to conflicts. Instead, we adopt a standard for ID generation, employing a prefix along with a local increasing ID to create a globally unique ID. For instance, "Dublin:10" indicates the datasource prefix "Dublin" and the local ID "10" within that source.

Sentiment Predictor Service (Machine Learning)

- **Objective**

  There are countless reviews worldwide, making it impractical to employ experts to identify their sentiment. Hence, we require a machine learning model to assist in this determination and relay the predictions to the subsequent process for calculating various movies' ratings.

- **Implementation**

  This service plays two roles within the Kafka system. It acts as a consumer, predicting the sentiment of reviews from the message queue, and as a producer, sending this sentiment back to the message queue for other services to calculate ratings.

  The predictor is built upon a pre-trained random forest and an encoder. The text undergoes encoding to convert it into a vector, which is then passed to the random forest to obtain a binary sentiment result (1 for positive, 0 for negative).

  The binary sentiment, along with the global review ID and movie ID, will then be sent back to the Kafka message queue under another topic named 'result' as the producer for subsequent calculations.

Movie Rating Service (Redis)

- **Objective:**

  To calculate the rating for each movie, we require both real-time sentiment and historical data. We determine the rating for each movie based on its positive sentiment rate, calculated as the ratio of the number of positive sentiments to the total reviews on that movie. We then need to store these two values for each movie and update them continuously upon receiving new review sentiments. Due to the heavy search and update operations involved, Redis is the optimal choice due to its high performance.

- **Implementation:**

  The Redis Service continuously listens to the 'result_topic' from the Kafka message queue. Upon receiving new sentiment data, the values for that movie are updated accordingly.

  Additionally, we have incorporated a visualization feature into this service to display the real-time rating trend of each movie.

MongoDB Service

- **Objective:**

  For fault tolerance, we need a database to store the data processed by the sentiment prediction service. When a moview review enters the topic and after the sentiment prediction service update the value, the results should be saved for further use.

- **Implementation:**

  When selecting the database source, we choose MongoDB as an example of NoSQL database, which is a more popular choice than MySQL or SQLite. It also provide scalability and flexibility to handle the real-time updated user input. We also use Springboot to build the service and thymeleaf for the front-end web development.

The MongoDB service will continuously listen to two topics. It will consume the review from the test_topic and parse the JSON message into each content and store them into the database, to maintain the rating by the sentiment prediction service, at this time the rating will be set to -1. The database servive will also listen to the result_topic whose messages contains the prediction and the review id. Since the review id is unique, the original revew can be found by using the query function from the MongoDB packages. Once the entry is located, Update method is used to update the rating colum with the prediction (0 for negative, 1 for positive)

To present the data better, front-end webpages are developed to display the database using web controllers and provide a search function that can search by movie id and get the amount of the reviews and the positive rating.

Benefits from Distributed System

Scalability

The scalability of our system can be discussed in two aspects: functionality and hardware.

- **Functionality Scaling:**
  In our system, developers don't need to worry about conflicts with existing code when developing new services, whether in the backend or frontend. This is because we employ a microservices architecture, and all microservices interact solely with the Kafka message queue. For interactions with the frontend, there is a uniform REST API to regulate actions. Therefore, we can develop new services even in different programming languages, which helps us achieve the optimal solution.
- **Hardware Scaling:**
  In infrastrcture level, all the technologies in our tech stack-Redis cluster, MongoDB, Kafka cluster, and Zookeeper cluster—are designed to support horizontal scaling, making scaling very straightforward. The number of consumer and producer of Kafka Message queue is not limited, which

make the service node very easy to scale up.

Fault Tolerance

In a data-driven system, faults may arise in message transfer (e.g., Kafka breakdown) and service disconnections.

When employing a Zookeeper cluster to coordinate the Kafka cluster, this infrastructure is highly robust. Following the principle of Zookeeper Atomic Broadcast, the system can perform well as long as no more than half of the Zookeeper nodes fail. With a valid Zookeeper cluster, the Kafka cluster remains operational until no Kafka nodes are available for use.

For service disconnection, Kafka is utilized to store data that has not been consumed. Additionally, MongoDB serves as a backup for Redis; in the event of a Redis crash, we can still calculate the latest ratings for each movie using MongoDB, as it contains all the necessary information.

Contribution

- **Tianhao Liu:** Data Collection Service + Sentiment Prediction Service
- **Haotian Shi:** MongoDB Service + Review Data Visualization
- **Xiaoyu Han:** Redis Service + Real-time Rating Visualization

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

- **Challenge**: we are using different operating system, portability of infrastructure may be a problem

  ```
  We use a lot of docker containers for infrastructure
  (redis/kafka/zookeeper/mongoDB)
  ```

- **Challenge**: One of the problems we faced when connecting the back-end with the front-end was the cross-origin link problem, which caused the front-end page to be blocked by CORS policy and could not get the value from the back-end through the URL.

  ```
  After trying several approaches and looking around, we found that setting rediscontroller's CrossOrigin to public could solve this problem.
  ```

What would you have done differently if you could start again?

For the distributed architecture, no, we have conducted extensive research, including industrial applications, and we believe this is the most effective and cost-efficient way to construct a distributed system.

But for the implementation of some specific service, yes. In the choice of model used for emotion analysis, we may try to use StanfordNPL for emotion analysis, this model can more accurately identify the type of emotion, such as very positive, feeling bland, or extreme disgust. In this way, we can get more accurate results when we calculate the popularity of movies in the future.

What have you learnt about the technologies you have used? Limitations? Benefits?

Microservice architecture provides us with numerous benefits, as we can develop the application in parallel.

Echart is a very handy data visualization plugin. It supports many types of data visualization and is easy to learn. During this use I learned how to retrieve the backend data from the url and update it in real time. One shortcoming I noticed in this project is that Echart loads slowly because it changes too quickly when the data is updated too often.