CS209A Final Project Report

冷子阳：12011513 [Backend]

唐昕宇：12011439 [Frontend]

1. Intro

For this project, we choose questdb and dubbo as our demo repos for their considerable scale to provide abundant data for analysis while not overwhelming the API or analysis structure. Questdb is an open-source time-series database for fast ingest and SQL queries implemented using java mainly while dubbo is a high-performance, java-based, open-source RPC framework, and we are intended to study their developing scheme by implementing some analysis according to their activities.

The whole project is a pipeline that can complete data collection, analysis, and serving with a simple run command. It consists of **Flyway** db immigration, data fetching & storaging, **mybatis** mapper, caching and api serving which each serves as an indispensable step for the final presenting.

2. DB Immigration & Data Fetching

For data persistency we are using **mysql** on docker for repo-related data & meta-data storaging. And to automate the whole process we use **Flyway** for database initiation and schema history management for following modifications to the database. For database structure, we design 5 entity tables, namely *repository*, *user*, *commit*, *issue,* and *release* to store the related information fetched from github api and 1 relation table *repo_contributor* to avoid the duplication of *user*.

The process of fetching data from github api consists of two separate steps. First we request with **okhttp** client and parse the response with *Object Mapper* that utilize the Bean with Json properties generated from any valid response from github api using the tools here, thus we will not lose any valid information and the related additional info can be retrieved whenever we need them. Then we use a repository factory that takes in the objects constructed from the last step and retrieves the useful fields to construct the object that we are storing into the database tables. All of the steps are implemented through *DataPreparer* works together with lists of dto classes generated.

Several highlights lie behind the implementation of this section, namely

1. Extracting the Flyway *migrate* process and using *@DependsOn* annotation to ensure the execution sequence of schema migration and data insertion and avoid conflict.
2. Using *@PostConstruct* annotation to automate the data fetch process by utilising the characteristics of Spring framework.
3. Using configuration properties to specify the repositories need to fetch and database connection info.
4. Using parallel stream and *@Async* function to increase the efficiency of requesting and parsing data.

3. Mapper & Service Implementation

For the operation between the program and database, we choose **mybatis** as the framework and use SQL to interact with the database to query, insert, join information to satisfy the needs for different forms of data for processing. The Java side mapper simply works as function call and package the passing parameter into fields that can be retrieved in SQLs defined in mapper *.xml* and execute database manipulation.

Service constructs of the foundation of our project, and apart from the regular analysis and statistics, we implement some services that may possess certain useful and profound results.

First is issue key words analysis according to the part of the speech, and the other is to predict the next release time and number of commits according to the history releases. These two features utilize the implementation of **stanford core nlp** and **apache commons math**.

While the process of natural language processing for the issue key words requires first tokenize the sentence, determine the posTag and then classify them according to their parts of speech, this can be time consuming and posing a negative effect on the interaction, we are using **ehcache** to cache the analysis results according to the function signature. Different from redis, ehcache fully utilizes the JVM memory and caches the returns of specified functions the first time it is invoked for every runtime. And the controller also has an api for active caching whenever a quick response is needed in the foreseeable future. This greatly reduces the response time and increases the overall performance. All of the services and their implementation and optimization lie in *RepositoryServiceImpl* class.

All of the analysis results are returned to front end using a serial of DTOs predefined according to the front-end requirements.

4. FrontEnd

We use **Vue3** as our frontend frame and the **Arco Design** component library to construct the webpage. For visual data analysis, we import **Echarts** and **Apexcharts**, two open-source chart libraries, creating different charts we need.

For each chart, we write a .vue file to encapsulate it:

1. For **the top 10 of the most active developers**: Use World Population Bar in Echarts as its template. The y-axis is the developers' avatars and names. The x-axis is the developers' commit times. Sorting the number of commits from highest to lowest, the comparison can be seen more visually through the bar chart.
2. For **open issues and closed issues**: Use Doughnut Pie Chart in Echarts as its template, which can see the ratio of open issues to closed issues more clearly.
3. For **issue resolution time's Five-number Summary**: Use Box Plot Chart in Apexcharts as its template. The box plot can directly represent the five values of min, q1, median, q3, and max.
4. For **issue resolution time distribution**: Use Basic Scatter Chart in Echarts as its template. The y-axis is the issue resolution time. The x-axis is an issue-creating time (judging from the time of the first issue created). Combined with the scatter plot, the variance can be made more apparent.
5. For **the top 20 of the issue keywords**: The word cloud is implemented through the Echarts tool, while the size of the word can reflect the frequency of the word.
6. For **commits made between each release**： Use Line Chart in Echarts as its template. The y-axis is the number of commits, the x-axis is the name of each release. The line chart can reflect the trend of numbers.
7. For **the time that developers made commits**: UsePunch Card of Github in Echarts as its template. This chart provides a visual representation of the period and day of the week for developer commits.

5. Insights

5.1 Developers

We have provided the total number of developers, as well as the names, avatars, and the number of commits of the ten most active developers. From these views, it is possible to visualize who is contributing more to this repository.

Improvement: We can add a hyperlink to the avatar to this developer's personal page, which is more interactive and allows users to get more information about the developer.

5.2 Issues

Firstly, through the ratio of open issues to closed issues, you can find out the issue resolution status of the current repository. For example, if there is a large percentage of open issues, it will tell users of the page that they may also encounter a lot of problems when using the current repository.

Then for the issue resolution time, The average value represents the most possible time to solve an issue. we also provide a box plot of min, q1, median, q3, and max values, to reflect the data's **dispersion**. This data can give users a reference, for example, if they have a problem when using this repository and need to raise an issue, how long it might take to get resolved. What's more, combined with the scatter chart with variance, users will understand the **stability** of the issue resolution times, which also for users as a reference that how badly their general estimate might fluctuate. And users can also visualize the peak of issues whose resolution times are long, and are raised.

The third main part of issues is we collect issues' titles, descriptions, and comments to analyze, and pick the top 20 most frequently occurring **verbs** and **nouns** respectively. For nouns, users may learn where the problems mainly occur, or the issues' main topics. And For verbs, users may learn what actions can cause problems.

Improvement: What could be improved is that all our variables are in hours, which is not intuitive when the time is too long. It would be better to show the months, weeks, days, hours, minutes and seconds spent. Another is that in the scatter chart, due to the large data disparity and the large y-axis coordinate units, the issues that were solved quickly are concentrated at the bottom of the chart and you cannot see exactly how long it took. It would be better to make a partial chart afterward. Last, the issue of text processing is relatively simple, after which more methods of nlp can be studied to filter useless text and try to extract more useful information for aggregation.

5.3 Releases and Commits

From the line chart that represents how many commits are made between each release, users can learn the **changing trend** of commit times. In addition to this, we have also made **predictions** for the next release, then provide the time of the next release, how many commits may be between the latest release and the next release, and the current progress as a reference. This allows the user to better understand the status of the current repository version update. For example, users can choose whether they will wait for the next release according to these data.

From the GitHub punch chart on our webpage, users can see what time of day all the commits are distributed, or how they are distributed during the week. They can learn when the developer team is possibly active.

Improvement: The current prediction method is very simple, the prediction can only be used as a reference if the prediction is not allowed to bring misleading. It is possible to make more rigorous predictions later, and at the same time increase the graph of data to bring more reference value to users.