Github Data Analysis

Wide Project Report

YUAN Yingzhe
Computer Science and Enigeering
HKUST
Hongkong, China

GUO Jiong
Computer Science and Enigeering
HKUST
Hongkong, China

***Abstract*—GitHub have huge user group. Every moment there are hundreds of activities happen. To analyze the GitHub data continually, we use some opensource big data component such as Hadoop, spark, hive to construct a GitHub data processing framework, which aims at handling massive activity records and analyze data to find out patterns of users, repositories and languages. Using python crawler to get data from GitHub. By different data processing skills, the data is transport by certain channel to provide the downstream usage. Building the data warehouse base on the structure and content of the data and make three layers of the data warehouse. By spark, making data analyzation and store the result in the relational database. Finally present the graph and table on the dashboard. The code can be found in *https://github.com/gjbang/msbd5003-project* .**

I. Introduction

GitHub is an open source platform for developers to share their codes and collaborate with others. Every day there are innumerable user activities happening and these are collected by the GitHub. The data contains varieties of activities on different repos. From these data, we can extract many extra information like the recent hottest language. One important things on this mission is the data storage and data process. Large data set will cause huge pressure on the traditional relational database. Therefore, we need to seek a different way to implement the data analyze system, which can afford high load, store huge data set and have high fault tolerance. For these purposes, we choose several appropriate methods and techniques, and successfully build a system of data warehouse to deal with high load data.

II. Dataset

All the data we will use in this project are from Github. There is not an existing complete dataset to get enough well-structured data for our project, so we need to collect data and preprocess by ourselves. In this section, we will illustrate why we choose Github's data and what the data sources are.

A. Why Github

Github is the largest developer community and the largest platform for developers to share their codes and collaborate with others in the world. By analyzing the data from Github, we can get some insights about the development of open source projects, the development of programming languages, and the development of the developer community.

B. Source I: Open API

Github provides a series of open API for developers to access most of the informations on Github. This 'Github REST API [2]' is mainly used for creating integrations, retrieve data, and automate workflows, but we can use it to query specific data needed for our project.

There are mainly two main areas of data:

- **Users:** Basic open informations of user, such as id, name, email, following users, repositories owned, etc.
- **repositories:** Basic information of public repositories, such as id, name, owner, description, stars etc.

We will provide some examples results of API query in AppendixA.

There are limits of current official REST API:

- Some need authentication, which means we need to provide a valid token to access the data.
- Some API cannot be called too frequently, which means we need to wait for a while before we can call it again.
- Most of API need an ***id*** as its query parameter, so we need to get the id first before we can query the data, which means we need to construct a massive id list.

After considering the above limitations, we decide to use the Github Archive [1] as our main data source and use the REST API to get some additional data or search some specific content.

C. Source II: Github Archive

GH Archive [1] is a project to record the public GitHub timeline, archive it, and make it easily accessible for further analysis. In fact, it also based on Github's Open API, while only use the **activity** API:

```
https://api.github.com/events
```

GH Archive will crawl github's activity data in real time and sort them as json file in order of when they occurred. Activity archives are available starting from 2011-02-12, and are updated amost half day, so the total size of data is **TB** level.

Considering too large size of the data and the the earlist data using a different format, we will only use the latest part of them. In fact, the current amount data of data in an hour is about 700MB, so the latest data is surely enough for our project. The corresponding compression packages with specific time can be downloaded easily, and the minimal time interval of packages is one hour.

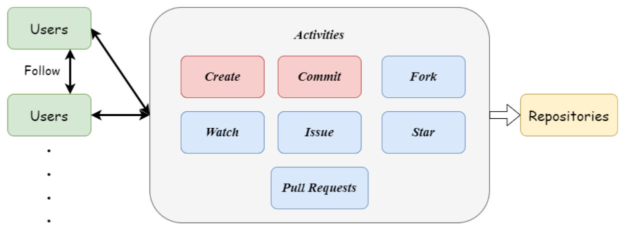

Fig. 1. This figure shows relation between user and repository and basic activities.

The archieved activity data contains almost all 20+ event types provided by Github, which ranges from new commits and fork events, to adding members to a project. We will only pay attention to some key event type, such as *commit*, *create*, etc.

We will give some example of the data format in AppendixB.

D. Summary

Based on almost all activity record, Github's Open API can help us to get almost all related information. For example, from **GH archive**, we know user a push a commit at 12:00 in repository A, then we can get almost all details of a and A by calling Github's API.

III. Task

Github has provided some basic data aggregation for user to explore. For example, in ***Trending [3]***, Github shows the most popular repositories in the specific time range, language and spoken language. In ***Topics [4]***, Github shows repositories' topics according to tags of repositories. However, these data analysis are very basic and not wide to cover most of the information in Github.

In this project, we will use spark to construct a real-time processing framework, which aims at handling massive activity records and analyze data to give a more comprehensive analysis of Github.

There are main tasks in this project:

A. Data Preprocess

1) **Collect** data. Crawl Github Archive [1] and use Github API [2] to jointly query more detailed information about activities according to indices in Github Archive. Though Github Archive update several times a day, we will simulate it to a real-time stream to process data in real-time.
2) **Preprocess** data by Spark. For complete each json record, split it into several parts, filter useful parts and store them in different tables.
3) Data **Persistence**. Design different schema and construct key meta tables to store necessary information, statistics result or some data not easily obtained directly.

B. Data Analysis

1) **Statistics**. We wiil try to construct a dashboard to show some basic statistics of Github, such as the repository with the highest frequency of commits, the most frequently commits time period, etc.

C. Data Visualization

1) **Dashboard**. We will try to construct a dashboard to watch the process of data process and show the results of data analysis by deploying a webpage.
2) **Interaction**. We will try to make the dashboard interactive, such as users can choose the time range of data to analyze, or choose the language to analyze, etc. Results of corresponding process not only comes from persistant database, but also from real-time stream produced by Spark.

IV. System Framework

A. Cloud Server Configuration

1) Memory and Cores: The system architecture contains many components. The metastore of hive is deployed on mysql, so that hive can be used by multiple users. Therefore, we need to choose a server with large memory to accommodate all components. Considering the system requirements, we choose a server with 4 cores and 16G memory.

2) Server Type: Our task is to build a data analysis service platform that continuously obtains log information and do data analyze jobs. We chose preemptible instances because of their relatively low price. And we have developed a set of fully automated scripts to deploy the cluster, so that if the server needs to be released unexpectedly, the service can be easily deployed to the newly created server.

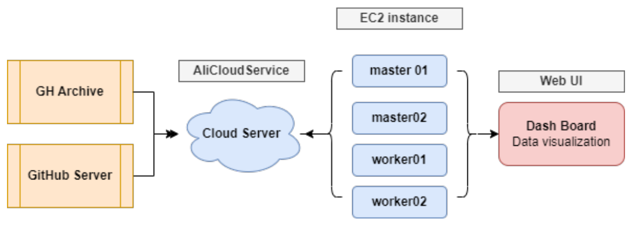

Fig. 2. A figure shows types of servers

3) Final Cluster Choice: For the final cluster, we use 4 server, which are named as *master01*, *master02*, *worker01* and *worker02*. Different server played as different character in the cluster.

The Server details information:

- Instance type: Preemptible Instance
- Specifications: 4v CPU 16 GiB (I/O Optimized)
- Storage: 40GiB Cloud disk for each server
- Network: 100Mbps

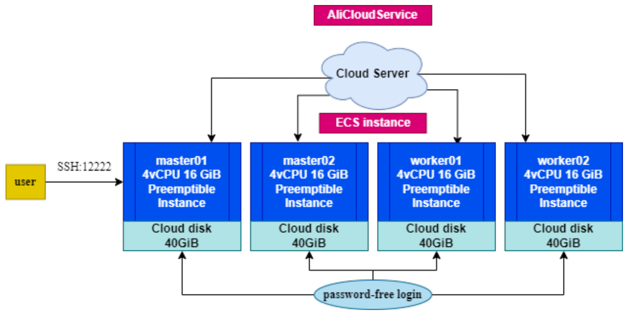
AliCloudService
Cloud Server
ECS instance
user
SSH:12222
master01 4vCPU 16 GiB Preemptible Instance
master02 4vCPU 16 GiB Preemptible Instance
worker01 4vCPU 16 GiB Preemptible Instance
worker02 4vCPU 16 GiB Preemptible Instance
Cloud disk 40GiB
password-free login

Fig. 3. A figure shows details configuration of clusters

4) Auto Configuration: Preemptive servers may be automatically released for lack of resources by Aliyun, so we need to purchase new computation instances, deploy the original big data system quickly and recover data collected when that worst situation happens. Based on this requirement, we have implemented a complete automatic configuration shell script, so that only one script need to be executed on any one single node to complete the environment configuration and deployment of the whole cluster, which greatly improves the efficiency of the cluster deployment and reduces the workload of the administrator.

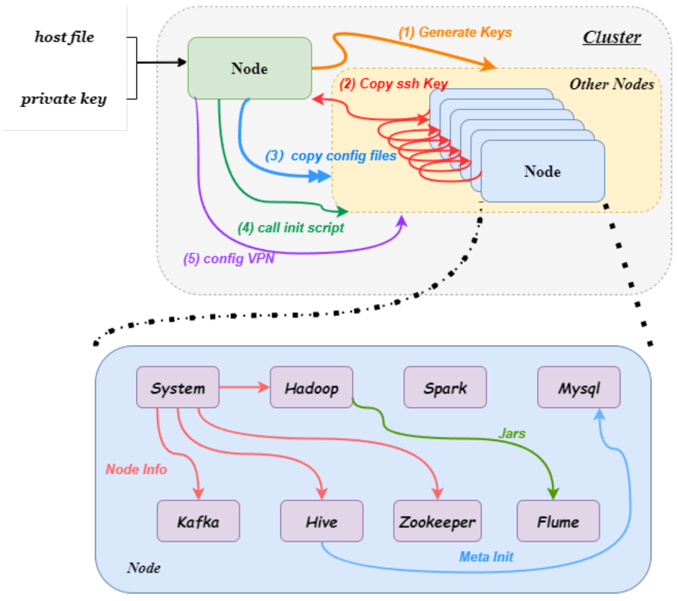

Fig. 4. A figure shows the control flow of auto configuration

Cluster: For the whole cluster:

1) Upload private key (for log in server initially) and host file (record the ip and name of all nodes) to any one nodes *A*.
2) Execute autoconfig shell script, which will do the following things one by one:
   a) Require each node to generate a new ssh key pair and update sshd configuration.
   b) Each node broadcasts its own public key to all other nodes for password-free login. Then only use the new own key to log in to other nodes after that.
   c) Synchronize all necessary config files of components(hadoop, spark, etc) from *A* to all other nodes.
   d) Execture the installation script on each node's background.
   e) After the installation is complete, all nodes need to config a VPN proxy to access the outer Internet for downloading Github data.

All big data components will be installed when installation script is executed. The installation script will do the following things one by one on each node:

1) Install system software, update system environment variables and create necessary directories.
2) Modify configuration files of each component according to the node's role and special information, such as zookeepers'id, etc.
3) Download and install all components one by one, and copy the configuration files to the corresponding location.
4) Copy essential ***Jars*** of hadoop and spark to other components' *lib* directory, such as flume, which is used for ensure consistence of the version of all components to avoid potential problems.
5) MySQL will be installed on *worker02*, modify access authority, and init database schema. The JDBC driver will be installed on all nodes.
6) Init hive metastore database on *worker02*. The hive schema will be stored in MySQL.

5) Security Control: After deployment, servers are frequently attacked from unknown sources, including DDoS, mining viruses, worms, and remote controlling, which causes the extremely high CPU usage, high disk IO, and unable to access the server. Killing processes, resetting servers, and recollecting data took up over 40% of our time.

Although it has been observed that the behavior after entering the server is to inject code into the process, it is not clear by what means the server was successfully accessed and permissions obtained. We have taken the following steps to try to avoid being attacked:

- **Change the default port of sshd**: The default port of sshd is 22, which is the most frequently attacked port. We changed the port to 2222, which greatly reduced the number of attacks.
- **Only allow strict key login**: Some instruction of automated configuration steps need to be provided password, we generated a random length of 40 characters password by openssl and only use it once. After the configuration is complete, the password will be deleted, and all communication will be based on the rsa key.
- **Close unnecessary ports**: We only preserve port for ssh login and data visualization for external network access, and other ports of big data components are only open to the internal network.
- **Install anti-virus software**: We installed *ClamAV* on each node to scan the system periodically.

All above measures cannot completely prevent attacks, and we believe there is an unknown vulnerability in linux system or big date components. The corresponding logs are in the Appendix C-C.

B. System Architecture

Our goal is to build a distributed file system and computing analysis system based on cloud servers. After identifying the servers and clusters, we need to decide where to deploy the components and how data will flow between the different components within the architecture.

1) Technology selection: Considering our needs, we need to build a distributed file system and distributed computing system for continuously input log information, and need to store related information such as users and warehouses. So we choose the most commonly used technology in the field of big data: Hadoop architecture. Considering the data form we need, we use hive to store the data as hive tables. For faster analysis, we use spark for distributed computing and analysis. According to the main thread, we add more components to it to maintain the continuous operation of the whole system.

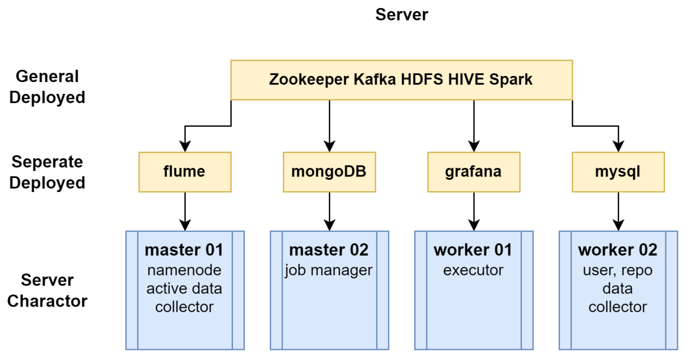

Fig. 5. A figure show the deployment of the system

2) Deployment: Balance the server workload so that the cluster can perform at its best. Therefore, it is important to plan the placement of components. Below are the locations of the different components on the server and their basic functionality.

For the entire analysis system we use the following components. Here we briefly introduce these components:

1) **Zookeeper**: ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.[1] https://zookeeper.apache.org/
2) **Kafka**: message queue, which temporarily stores data and sends it downstream. It is the data source and key part of subsequent data stream processing.
3) **Hadoop**: Hadoop includes HDFS, Yarn and MapReduce. HDFS is the main storage place where we store data. HDFS is deployed on each server. Use Yarn as a resource manager. The resource manager is deployed on master02.
4) **Hive**: Hive data is stored in HDFS. The hive engine is very inefficient, so only use hive as storage and data management on HDFS. Hive is deployed on the entire cluster.
5) **Spark**: Spark is deployed on the entire cluster as the main component for data processing and data analysis. Also use **Spark Streaming** to process real-time data streams.
6) **MySQL**: The data of users and warehouses are stored in MySQL, and MySQL is deployed on worker02. At the same time, the hive meta store is also on MySQL.
7) **Flume**: Track the data in the data capture folder and transfer it to Kafka. It will also transfer data from Kafka into a temporary folder in HDFS. The data capture process is deployed on worker02.
8) **MongoDB**: MongoDB is deployed on master02 for expose the API. The data is stored in json format, which facilitates a background application such as flask to respond to web requests. We have stored analysis results in MongoDB.
9) **Grafana**: Grafana is deployed on worker01. It's the interface of data visualization. It can connect to MySQL to get data and display it in the form of charts. In fact, it get well-structure data by constructing SQL statements and then send it to MySQL for processing. It's easy to combine single figure into a dashboard. Grafana also provides port mapping for external network access.

The version of key components will be listed in the AppendeixC-A.

C. Data Flow

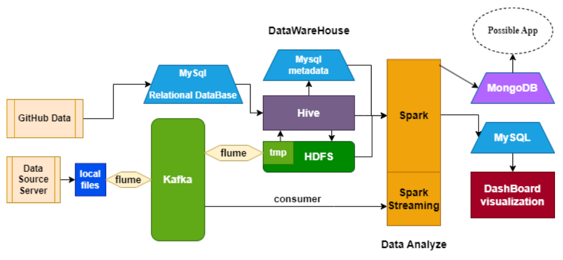

Fig. 6. A figure shows the data flow of the system

The server structure section introduces the general data flow direction, and here is a detailed description of the data flow in the overall analysis system.

1. Data source: The data sources are divided into two parts. The first part is GitHub user behavior log information. The source of this part of the information is the GitHub Archive website, from which one hour of data can be obtained every hour. We temporarily store the acquired data in a folder of worker02 through the Python program for subsequent data capture. The second part is GitHub user personal information and repo information. The source of this part of information is the official server API provided by GitHub. We use Python crawlers to obtain user and repo data through these APIs and write them into the MySQL database, and the database keeps updating these data.

2. Data transmission channel: Data transmission channels mainly include flume and kafka. The obtained log information tracks the file information through Flume, and transmits the continuously arriving data content to Kafka. Kafka internal data can be used by spark streaming through consumer. At the same time, Kafka and HDFS are connected through flume. The data accumulated in Kafka is consumed in batches and stored in the temporary folder of HDFS.

3. Data load: MySQL memory has continuously updated user and repository information. The temporary folder of HDFS has continuously added log information. We incrementally synchronize MySQL data to Hive through the spark program. Build it into a Hive table. The data of the Hive table is stored in HDFS. At the same time, the log information in the temporary folder is imported through Hive, and the original data is transferred to Hive for subsequent use by using the JSON serde parsing package.

4. Data Warehouse Construction: Build a data warehouse based on the acquired data. The details will be explained later.

5. Data analysis: Spark directly extracts the data in the Hive data warehouse to analyze various indicators through the built data warehouse, and writes the analyzed results into multiple different databases for different purposes: Write to MySQL for subsequent data display. Write to MongoDB to expose Api externally. Write to Hive for historical data storage. Spark streaming directly obtains data through kafka for real-time data processing, so as to obtain updated and fast real-time data display.

D. Data Warehouse Construction

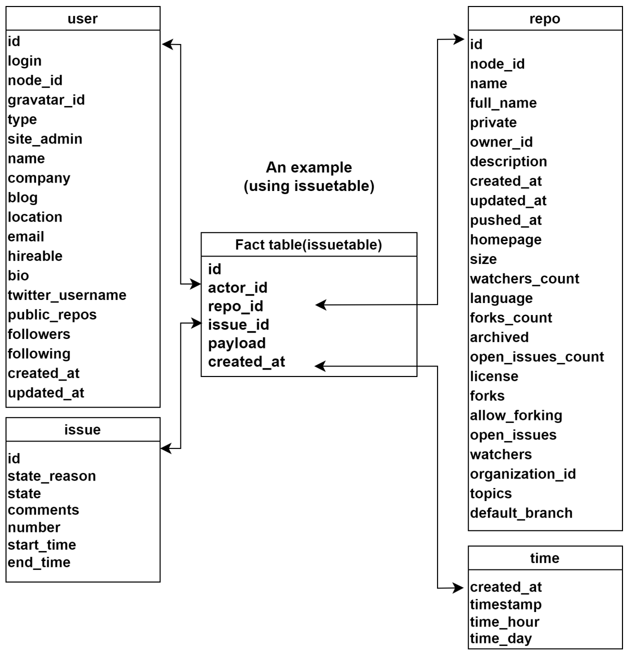

Fig. 7. A figure shows the schema of the data warehouse

1) Modeling: The basic structure of data warehouse modeling adopts constellation model. There is a star schema for each transaction table, and different transaction tables share the same dimension table to form the entire constellation model.

Data granularity selection: The log data itself comes from every operation of GitHub users, and there is a corresponding time record for each operation. Since the data is acquired in one batch at the same time during the data acquisition process, the time granularity is selected to be as fine as 1 hour.

Determination of dimension table and transaction table: the dimension table adopts four dimensions of user, repo, issue and time, and the transaction table is divided into 8 types of transaction tables, such as create, delete, pull, and push, according to the different behavior of users.

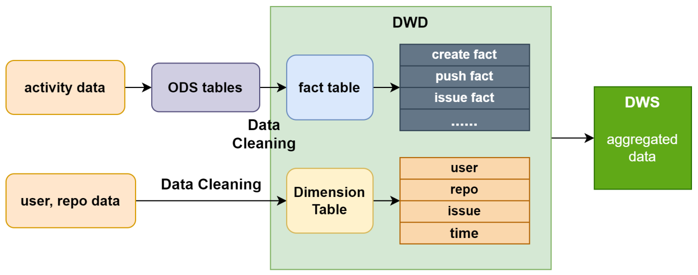

Fig. 8. A figure shows details of the data warehouse

2) Hierarchy:

- **ODS layer**: When data is imported into Hive through the HDFS temporary data storage folder, the initial ODS layer table is formed. The ODS table stores the initial data without any data cleaning and processing.
- **DWD layer**: The data of the DWD layer comes from ODS and MySQL. The first is the transaction table. The data in the ODS layer is cleaned and preprocessed through the Spark program, invalid data and useless attributes are removed, valid information is screened, the retained content is determined, and the data is sorted out. Finally, different styles of data tables are constructed according to the type of data. These tables are all derived from the original user activity data of the ODS layer and the transaction tables are finally constructed. Then there are dimension tables. The specific information of the dimension table has already been determined, and there is synchronously updated data in MySQL. For the synchronization update process of MySQL, incremental synchronization will be performed in Hive to import data. Therefore, historical information of users and repo data will also be stored in Hive.
- **DWS layer**: Aggregate data at different levels for existing historical data information. After data aggregation for different time spans, the snapshot information is saved in the Hive data table to form various basic indicator results.
- **Data display**: Spark will use these basic indicators for analysis and processing, and transfer the results to the

MySQL database for data display.

V. Data Analysis

After the data was aggregated into the warehouse, we collected various tables with good structural characteristics, and based on this, we used spark to further get some useful analysis results, which will also be stored into MySQL and MongoDB for further use.

Because user data on github is sparse, which means activity is not continuous and there isn't a strong pattern of activity, so there is no obvious causal relationship between attributes. For example, a large number of followers does not mean that he has a popular warehouse with a large number of stars, nor does the number of starts in a warehouse increase with the increase of fork. In other words, they reflect the same trend of users and warehouses - Degree of popularity, so we need huge dataset to analyze a very long time period to find a hidden weak pattern. Based on this, we only finished some simple aggregation and visualization of the data.

A. Aggregation

Based the action tables stored in Hive, we can use spark to aggregate thses data and feed results into MySQL.

Here are aggregation result tables:

Table Name	Description
EventAllCount	Count # of all event counts
userCount	Count # of users with timestamp
reposCount	Count # of repo with timestamp
issueBasic	Count # of various issues
issueInterval	Pivot # of issues with open interval
issueComments	Pivot # of issues as # of comments
issueNumber	Pivot repo with different # of issues
topLanguage	Top active languages each 10 minutes
topActiveRepo	Top active repos each 10 minutes
topActiveUser	Top active users each 10 minuts
topActiveRegion	Top active region each 10 minuts
topActiveCompany	Top active companies each 10 minutes
realTimeCount	Count # of events each 2 minutes

Except for *EventAllCount*, all tables will use **append** mode, so there are time series for each attributes in these tables, which can reflect trends of the attributes. Also, we can make more coarse-grained aggregation based these fine-grained tables.

B. Visualization

We used grafana to visualize the results stored in Mysql, which are produced by spark.

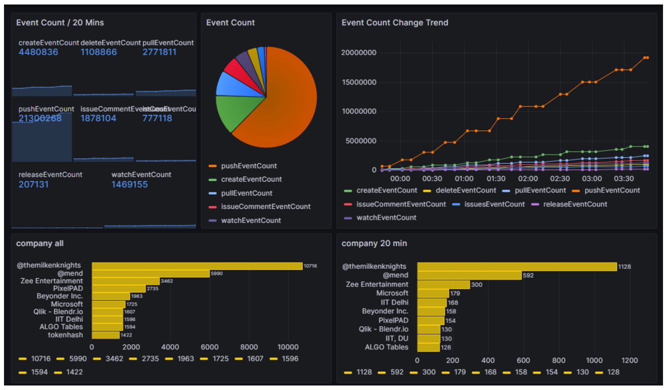

Fig. 9. Grafana Dashboard

Figure9 shows count of all kinds of event collected in Hive, ratio of different kinds of event, the trend of change of all kinds of events, the top company in the last 10 minutes, and the top company in the whole hive database.

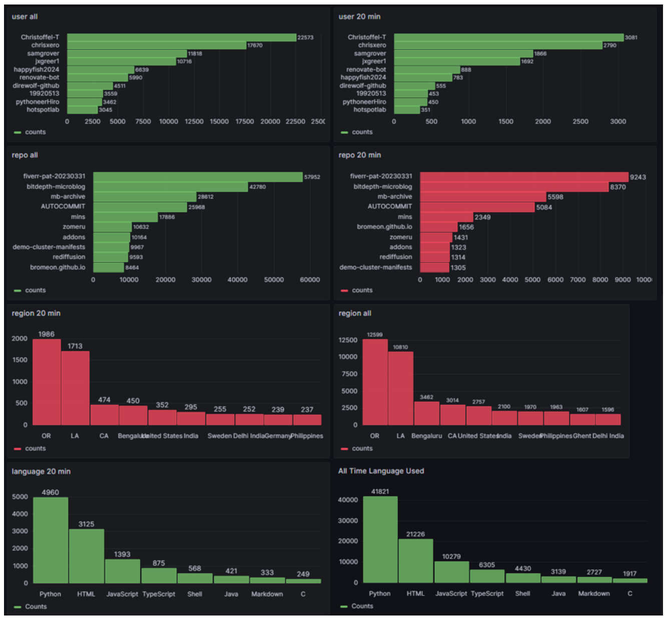

Fig. 10. Grafana Dashboard

Figure10 show the count of users' event in the whole hive database and in the last past 20 minutes, count of repos' event in the whole hive database and in the last past 20 minutes, the top region in the last 10 minutes, and the top region in the whole hive database

There are indeed some abnormal user and repo with lots of commit in a very short time.

VI. Summary

Based on Github Open API and GH archive dataset, we implement a data processing framework to analyze the Github data. We use python crawler to get data from Github and store them in HDFS. Then we use Hive to convert data in well-structure form and use Spark to analyze the data and store the result in MySQL. Finally, we use Grafana to visualize the result in MySQL. This is a complete big data system containing data collection, data warehouse, data processing, and data analysis.

References

[1] https://www.gharchive.org/
[2] https://docs.github.com/en/rest?apiVersion=2022-11-28
[3] https://github.com/trending
[4] https://github.com/topics

Appendix A
GitHub REST API Example

There are some API's calling examples.

A. *Get All Followers of a User*

```
https://api.github.com/users/mateiz/followers
```

```
[
    {
        "login": "bennettandrews",
        "id": 1143,
        "node_id": "MDQ6VXNlcjExNDM=",
        "gravatar_id": "",
        "html_url": "https://github.com/bennettandrews",
        ...
    },
    ...
]
```

B. *Get Repository's Detailed Information*

```
https://api.github.com/repos/apache/spark
```

```
{
    "id": 17165658,
    "node_id": "MDEwOlJlcG9zaXRvcnkxNzE2NTY1OA==",
    "name": "spark",
    "full_name": "apache/spark",
    "private": false,
    "owner": {
        "login": "apache",
        "id": 47359,
        ...
    }
    ...
}
```

C. *Get All Commits of a Repository*

```
https://api.github.com/repos/apache/spark/commits
```

```
[
    {
        "sha": "0fde146e8676ab9a4aeafebb1684eb7a44660524",

        "commit": {
            "author": {
                "name": "Juliusz Sompolski",
                "email": "julek@databricks.com",
                "date": "2023-03-24T01:56:26Z"
            },
            "committer": {
                "name": "Hyukjin Kwon",
                "email": "gurwls223@apache.org",
                "date": "2023-03-24T01:56:26Z"
            },
            "message": "...",

            ...
        },
        "author": {
            "login": "juliuszsompolski",
            "id": 25019163,
            "node_id": "MDQ6VXNlcjI1MDE5MTYz",
            ...
        },
        "committer": {
            "login": "HyukjinKwon",
            "id": 6477701,
            ...
        },
        "parents": [{}]
    },
    ...
]
```

Appendix B
GH Archive

One single json record from the GH Archive.

```
[
    {
        "id": "2489651051",
        "type": "PushEvent",
        "actor": {
            "id": 3854017,
            "login": "rspt",
            "gravatar_id": "",
            "url": "https://api.github.com/users/rspt",
        },
        "repo": {
            "id": 28671719,
            "name": "rspt/rspt-theme",
            "url": "https://api.github.com/repos/rspt/rspt-theme"
        },
        "payload": {
            "push_id": 536863970,
            "size": 1,
            "distinct_size": 1,
            "ref": "refs/heads/master",
            "head": "6b089eb4a43f728f0a594388092f480f2ecacfcd",
            "before": "437c03652caa0bc4a7554b18d5c0a394c2f3d326",
            "commits": [
                {
                    "message": "Fix main header height on mobile",
                    "distinct": true
                }
            ]
        },
        "public": true,
        "created_at": "2015-01-01T15:00:01Z"
    },
    ...
]
```

Appendix C
System Configuaration

A. Version of Components

Component	Version
Spark	3.3.2
Hadoop	3.3.2
Zookeeper	3.7.1
Scala	2.12
Kafka	3.4.0
Flume	1.11
Hive	3.1.2
MySQL	0.8.24-1
JDBC	8.0.32-1

B. Ports

Some ports need to be opened for communication between components. Most of them only need to be allowed in a internal network, and only some Web UI and ssh need to be opened to the public network.

Port	Service	Description
8088	YARN	YARN ResourceManager
9870	HDFS	HDFS NameNode
9868	HDFS	HDFS SecondaryNameNode
9864	HDFS	HDFS DataNode
8042	YARN	YARN NodeManager
8090	Spark	Spark Master WebUI
8091	Spark	Spark Worker WebUI
18080	Spark	Spark History Server
9083	Hive	Hive Metastore
10000	Hive	HiveServer2
3306	MySQL	MySQL Server
8020	HDFS	
37017	MongoDB	MongoDB Server
3000	Grafana	Grafana Server
12222	SSH	SSH Port
2888	Zookeeper	Zookeeper Server
3888	Zookeeper	
9092	Kafka	Kafka Server
2181	Kafka	zookeeper and kafka
7890	Clash	VPN Server
7891	Clash	VPN Server

C. Security Risk Log

There are some basic characteristics of server after being attacked. The following log and statistical data can verify that we are really attacked frequently.

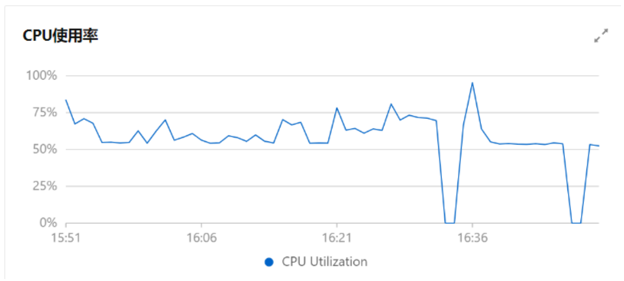

Fig. 11. Spark tasks are periodic tasks that are executed intermittently and have low load. After being attacked, viruses or mining programs usually occupy 50% of the core, so the cpu utilization is stable at more than 50%;

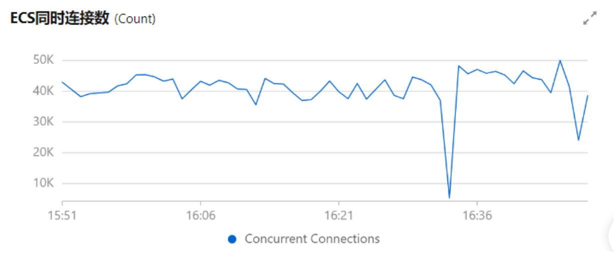

Fig. 12. Single normal user only has 20-30 ssh connections. After being attacked, the number of connections will soar to 40,000-50,000, resulting in almost unable to connect to the server to deal with the virus; The trough in the figure was caused by a server restart, after which the number of connections quickly recovered to more than 40,000

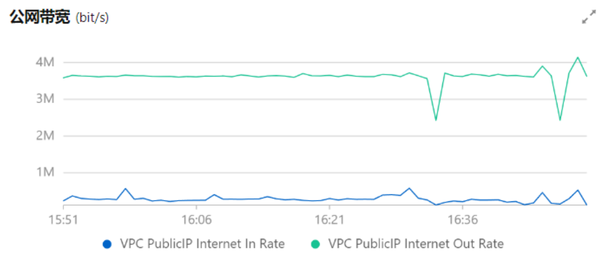

Fig. 13. Some mining viruses continue to generate data traffic in exchange for data

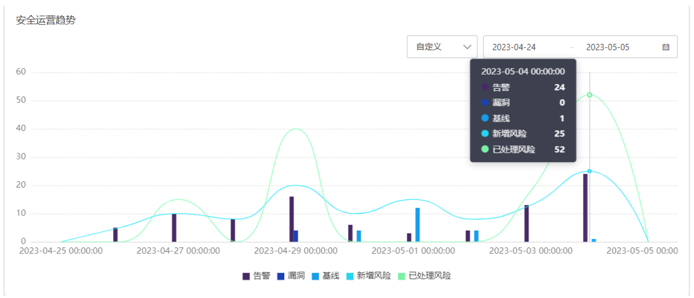

Fig. 14. Four servers in our cluster were sucessfully attacted every 12 hours. Each warning is a successful attack in the above figure.

On the presentation day of the project, which is 4th May, the server suffered the most serious intrusion, and the permission of MySQL stored key data was tampered, which resulted in the **loss of meta schema data of hive**. As a result, the data in the entire hdfs could not be properly identified. We had to delete all data collected, and recollect data again to complete the evening's presentation.