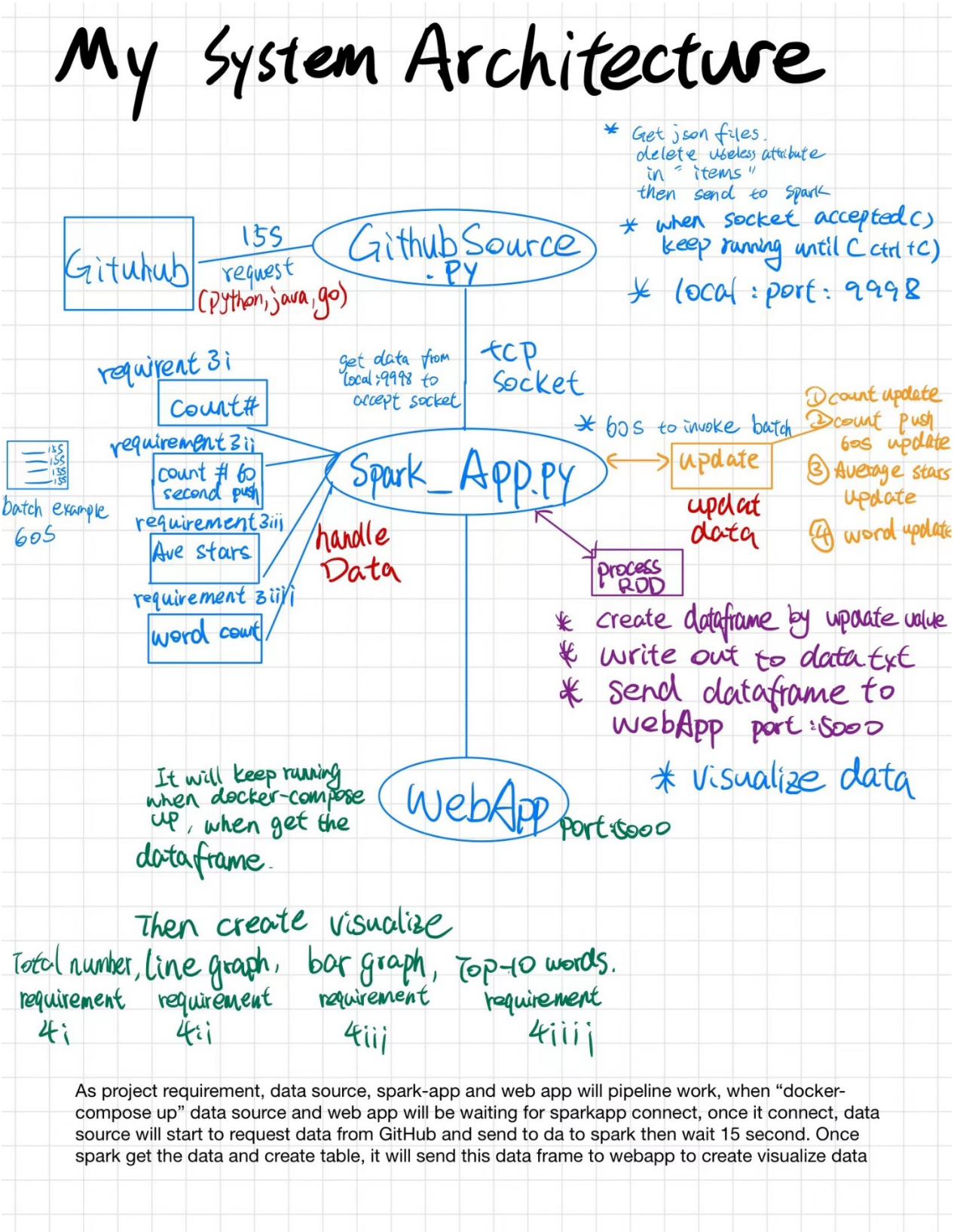
Student Name: Changhong He, 215873201



As project requirement, data source, spark-app and web app will pipeline work, when "docker-compose up" data source and web app will be waiting for sparkapp connect, once it connect, data source will start to request data from GitHub and send to da to spark then wait 15 second. Once spark get the data and create table, it will send this data frame to webapp to create visualize data

system architecture

I draw this project system architecture above because this is a pipeline job, then “GithubSource”, “Spark_App” and “WebApp” will run at the same time. This project has three main python files. The first one is “GithubSource.py”, These python files will create a TCP socket and set the port 9998, it will wait for Spark_App to accept transform. After Spark accepts, it will send a request to GitHubApi to get the data information. Then, we could delete useless attributes and only keep the attribute that we want “Items” list, “full_name”, ‘stargazers_count’ and “description”. Next, it will use a TCP socket to transform data to the Spark with port 9998. The second file is the “Spark_App” python file, it will accept to get the data from port 9998 and then set the batch interval time to 60, which means every 60 seconds to put the data into the batch and then manipulate data in the batch. We will find the repository count, count of “push_at” in 60 seconds, average_star, and top 10 words for each language description. Then create a data frame to show it in the terminal. Next, it will send to data frame to the “WebApp” with port:5000. In “WebApp”, it will get the data frame and transform it into a list. Then we could use the element in the list to create real-time information with each language's number of repositories, the number of repositories push_at in the last 60 seconds, the average star of the language, and the top-10 words for each language, then it will show in the website: local ip:5000.

Service Interaction

In the beginning, the user will use “docker-compose up” to start our services, at this time, all of the services will start ready to run. My Github source service will wait for my spark to connect, and my web app service will wait for the spark to send data. Once user inputs command “docker exec streaming_spark_1 spark-submit /streaming/spark_app.py”, spark service will run. Spark service will accept the TCP socket and connect it, then the Github source service will send the request to GitHub to get the data and send the clean data to the Port:9998 every 15 seconds. Spark service will set the batch interval time to 60 seconds, which means that every 60 seconds, Spark Service will deal with batch data. After Spark use batch data to finish requirement 3 in the project description, it will create a data frame by “Process_rdd”, and it will show the table in the terminal, and then send the data to port 5000 to let WebApp get the data. When WebApp Service gets the signal from the Spark Service, it will handle this data frame and change it to a list. Then we could discover the Requirement 3 data information in the list, the WebApp service will manipulate this data to create a real-time table and data on the Website: local-ip:5000. And then wait for Spark Service to send other data.

This is a pipeline job, it means that three services will run at the same time and they will interact with each other.

Spark application processes the streaming data

Hint: the style of the data is :{“LangaugeName”: [{},{},{},{},{}…]}

Firstly, Spark Application will create Data_Source_Ip and Data_Source_Port, it will set the streaming context to 60 seconds, which means Spark App will run each batch every 60 seconds. Then, Spark App will use socketTextStream to get the data from the 60-second batch. It will use the Map function to let each dictionary in the batch data

be a JSON style. And then do our first requirement3.1, it will use the map function to map the data, we could use lambda to check which language it is in the batch data, and then create a tuple that ("LanguageName", length of the List) to get the count of the repository in this language and then use reducebykey to plus the number of the repository with same language name. For then requirement 3.2, we could use window function, create a window 60 second, and each 60 second to get the last 60 second's rdd, then we could use function to compare the push_at time, then if the push_at time change during the last 60 second, then the count of the push_at time will plus one. Then we use reducebtkeyandwindow to reduce the key with same language type name to get the total count of the push_at time count. Then we start to do requirement3.3, it will also map the data to ("langaugeName", (total star of this data, count of the repository), then we could use reducebykey to get the total number of to let same language name's value to plus together, then we could get total star of the language and the total number of the repository, then we could use the map function to use star divide count to get our average stars.

```
def comparetime(data):
    #data is a list
    count = 0
    if int(len(data)) > 0:
        for itemsdict in data:
            for key in itemsdict:
                if key == 'pushed_at':
                    repopushtime = itemsdict['pushed_at']
                    repopushtime = repopushtime.replace('-', '')
                    repopushtime = repopushtime.replace(':', '')
                    repopushtime = repopushtime.replace('T', '')
                    repopushtime = repopushtime.replace('Z', '')
                    nowtime = rightnowtime()
                    FMT = '%Y%m%d%H%M%S'
                    d1 = datetime.strptime(nowtime, FMT)
                    d2 = datetime.strptime(repopushtime, FMT)
                    d1_ts = time.mktime(d1.timetuple())
                    d2_ts = time.mktime(d2.timetuple())
                    seconddifference = int(d1_ts-d2_ts)
                    if (seconddifference <= 60):
                        count = count + 1
        return count
    else:
        return 0
```

Last requirement 3.4, we could use the map function to get this ("Language_Name", [(x, 1), (y, 1) …]) and then keep using the map function to get this ([(language_Name!=x, 1)((language_Name!=y, 1)…]) which means it will join the language name and word name with "!=", then it will become the normal word-count problem, use reducebykey to get the result, and then use the map function to split language name and word by "!=", and then get our final style language name [(x,10), (y, 12)…]. Then we will use updatestatebykey to update our information, it will keep adding new information to our old information. ((Average Star will oldaverage star * old count + new average star * new count) / total counts). After that, we could get four datastreame with the right now value, then use UpdateStateByKey to upload our new

value to our old value. For requirement3.1 and 3.2, it will just count them together for aggregate, for reuirement 3.3, it will use (old_average_star * old total_count + new_average_star * new_total count) / (old_total_count + new_total_count), then we could aggregate new average star for all batch. For requirement 3.4, the aggregate function will record new value list with total word pair, and then use for-loop to check if the word pair exist in the old word pair, if exist then let the old_word_pair value + new word_pair_value, if not, then create a new word_pair in old_total_word_pair, then return the sorted old_total_word_pari. Then we could use [:10] to select the top-10 words. Finally, I will use the join function to let four datastream to join together since they have the same language name as the key. Then use foreachRDD(process_new_rdd) and use our new datastream to create a table and send it to the WebApp Service.