**RATE LIMITER**

**Version:1.0 R1**

**Submitted By:**

**Akash Gupta 8764098777 aksgupta3697@gmail.com**

**Problem Statement**

Identify a Developer’s profile on GitHub, when following attributes are provided:

1. First Name

2. Last Name

3. Location

Once the profile has been identified, extract following attributes from the profile:

1. Public profile details

2. Name of repositories where they are contributing

3. No. of commits made by that developer on those repositories

The solution should be able to take a list of developer information, rather than just 1 developer at a time. The data should be gathered in the minimum amount of time without wasting resources or violating any terms & conditions mentioned by the host. GitHub applies rate limits to the use their API, this rate limit differs between an anonymous user and a logged in user. The rate limits are in place to avoid any type of D/DOS attacks at the same time to avoid malicious scanning of user data. This rate limit needs to be honoured to avoid account lockouts, this could also lead to the IP getting flagged and blocked by the service provider. Hence we need to gather the data in a way that application operates in the given rate limits as specified by the host. You solution should be able to extract data of more than 1000 developers without breaking rate limits of github.

**1. Understanding API**

API is Application Programming Interface which allows two programs to communicate witheach other. Through API our project can use the resources of another application or website. We are using this API as a means to extract the data from the website by using multiple queries. There are two available forms, v3 and v4, where v3 is json and v4 is GraphQL output format specific. Both have different query styles. Json output has lesser end points hence requires a lot of cleaning to get the required data whereas GraphQL has relatively difficult to execute queries, but with more end points, is easier to extract specific data.

**2. Understanding API Key**

API is a unique key/token assigned to a user of a website to authenticate him/her to use its API. Github provides a token for their authenticated users. This key can then be used via 3 methods:

>OAuth2 : Basic Authentication

Ex: curl -u "username" https://api.github.com

>OAuth2 : sent in header

Ex: curl -H "Authorization: token OAUTH-TOKEN" https://api.github.com

>OAuth2 : sent as parameter

Ex: curl https://api.github.com/?access\_token=OAUTH-TOKEN

We have used the parameter authentication method.

**3. Rate Limits of Github**

Github has a rate limit to use its API. The rate limit is different for anonymous and logged in user. The rate limit of anonymous user on github can be checked from the following link <https://api.github.com/rate_limit> . To check the rate limit of a logged in user you have to provide the access token of the user in the link as well. For example: [https://api.github.com/rate\_limit?access\_token=”your access token”](https://api.github.com/rate_limit?access_token=) .

The search rate limit and core resources rate limit of anonymous user is 10 and 60 per hour respectively and of logged in user is 30 and 5000 per hour respectively.

**4. Approach**

The core search limit of Github is 5000 per hour per tokens. We are using 3 token keys to get the best efficiency of the project.

The primary problem for us was to understand and experiment with how to query data, in v3 and v4 of Github and generated 3 API key tokens from each of our accounts.

We interchange these keys when the rate limit of the key in use reaches 0. And if all 3 have reached 0, we put our system on sleep, until earliest of the three keys get recharged.

We initially used urllib library of python and developed out whole solution according to that within 2 days. But this solution was very slow and we weren’t even able to reach 3000 requests in one hour. We then tried multi-threading but python’s ‘Threading’ module has certain drawbacks which did not help us get much time saved in the request sending and honouring part. A request in github takes approx. 1.2 seconds (on Thapar wifi) to get honoured and replied to. Adding to that the storage and json cleaning, it was also a useless approach, and we ditched that code too.

We then used curl with ‘start /B’, which basically backgrounds the process on windows terminals, similar to the ‘&’ in linux. This allowed us to send hundreds of requests in one go. Because of extensive bottlenecks on the IO buffer and the file buffers, we had to tone down the frequency of the requests being sent by batching, which also allowed us to perform checks if the token had expired or not. We used the comfortable urllib to check for token rate limit depletion.

We also used a workaround for the search API. We have used legacy search instead of the normal search queries. Since even on an authenticated request, we get a maximum of 30 search requests, using legacy search was almost magical because it uses core requests too. So we were able to speed up our process a lot. The program works amazingly fast( compared to our initial approach). But there are some fallbacks on the accuracy because of this, like, exception handling for all edge cases is difficult, not many were covered up and also creation of a routine to handle the missed requests(given the information we have now as compared to the day we received the project, we can certainly accomplish it now, but for now we have saved the names we missed out, to be optimized on another day).

Due to excessive time spent in learning, and due to exams, not much was possible to be done on the missed out values part.

**5. Assumptions**

Constraint:- If because of some reason , your program stops in between of execution then re run the program by deleting all the files and output generated by the former.

Constraint:- Internet speed of greater than 1MBPS is needed for lossless and efficient data mining.

The entire application will have to be closed first(to release any occupied file buffers just in case)

Then all files and folders made by the program will have to be deleted, namely:

Folders : usernamesJson , repos , nocdata

Files : issues.txt , pdata.txt , outp.txt , rep\_list.txt

The program has been made with the assumption that the number of input values, i.e. combinations of first name, last name and location, will never be more than 14,670. In case it is, please break the input data into 2 or more csv files before using the program. Even then use the files separately, one by one, instead of together. (The 330 is a margin, 110 for each token, since the batch size is of 100, that means 10 margin for each token, just in case the token still is needed for confirming something).

**6. Suggestion of Additional Parameters for Optimisation**

In addition to firstname , lastname and location , we can also provide the id or username to uniquely identify the user. Id and username of a user are unique and the time that is being wasted in finding the corresponding usernames to the given combinations of (firstname, lastname, location) will be saved.

Location as a parameter is not even that necessary as we still got several usernames with the same name and location specified.

ID and usernames are unique on github and using the legacy search API, we can easily get all the information required by lesser depletion of available request limits.

**7. Improvements if more time were available**

There are 10+ ideas of improvements if more time was available, but to list a few:

We could incorporate multi-processing into the program to boost the rate of requests being sent.

The program currently takes less than 2 minutes to execute for data of 13 developers.

The program is still managing internal json cleaning and request data fetching at different times, this could be parallelized instead.

Exception management can be done better.

Missed out cases can be re-evaluated(if they downloaded after our file buffer started working then they might just return values now.)

With our current knowledge, we have already used the loophole of using legacy search of github instead of normal search but we could also simply redirect the requests to several servers, or even with ip spoofing, or even distributed systems, to deliver multiple requests together and redirected to one single local server in the end, but we lack the materialistic resources for that.

And so on…

**8. Execution Flow**

1. The input csv file is read and the entire data is converted into query request parts.
2. The legacy search query request is created and within the constraints specified, all queries are hit in a parallel manner. This floods the network but a time.sleep() function helps while the data is being written so as to ensure that the IO buffers and the file buffers are empty before next step.
3. The usernames are then read in batches of 100 queries and the repository data is pulled from the api.github.com, and within these batches, the json is converted into lists of data in the temporary folders with the data stored like Firstname\_Lastname\_Location\_Username\_RepositoryName.
4. After all the repository data has been obtained, batches of 100 are also created of these repositories, to hit api.github.com with queries to get the number of commits made by that user on that repository. The json data obtained here is also managed parallelly.
5. After this is finished, the data is arranged into a text file and then the missed out usernames, repository data is managed(We could not perfect this because we were too close to the deadline).
6. Clean slate protocol, to delete all the unnecessary files and folders.