Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16hp | Computer Science
Spring term 2022 | LIU-IDA/LITH-EX-G--22/055—SE

Data mining historical insights for a software keyword from GitHub and Libraries.io; GraphQL

Gustaf Bodemar

Tutor: Peter Dalenius
Examiner: Jody Foo

LINKÖPINGS UNIVERSITET

Data mining historical insights for a software keyword from GitHub and Libraries.io; GraphQL

Gustaf Bodemar
Bachelor thesis
Linköping, Sweden
gusbo010@student.liu.se

ABSTRACT

This paper explores an approach to extracting historical insights into a software keyword by data mining GitHub and Libraries.io. We test our method using the keyword *GraphQL* to see what insights we can gain. We managed to plot several timelines of how repositories and software libraries related to our keyword were created over time. We could also do a rudimentary analysis of how active said items were. We also extracted programing language data associated with each repository and library from GitHub and Libraries.io. With this data, we could, at worst, correlate which programming languages were associated with each item or, in the best case, predict what implementations of GraphQL they used. We found through our attempt many problems and caveats that needed to be dealt with but still concluded that extracting historical insights by data mining GitHub and Libraries.io is worthwhile.

INTRODUCTION

Extracting historical insights for software keywords can be beneficial in multiple cases, from visualizing popularity changes of a software library to showing the adoption of some software paradigm. Methodologies attempting this can differ both in scope and granularity. We are in this study interested in exploring one of these approaches. Our approach aims to create an inexpensive, rudimentary, yet representative historical overview of a software subject using keyword search. We will implement a program that data mines GitHub and Libraries.io for data to later analyze and visualize. Our ambition is to gain insights that can be helpful to developers and market analysts. We will test our program with the keyword *GraphQL* to create an example of what insights we can gain. We will also describe the challenges, possibilities, and opportunities we detected when data mining and analyzing GitHub and Libraries.io data.

This paper is relevant to two different groups. One where the reader is interested in understanding what historical insights may be achievable using this study's method to data mining GitHub and Libraries.io data, where the results we present showcase examples of what is possible to achieve using our method. Or a second group interested in the insights into *GraphQL* we gained using our approach.

GraphQL, our test subject, is an API query and manipulation language[1]. The GraphQL language specification defines, among other things, how a client can communicate with a server's API. One thing that makes GraphQL unique compared to parallel API methodologies, for instance, REST, is that it gives more freedom to the clients to define the structure of the server's responses.The GraphQL specification only defines the communication structure and thus allows various network endpoints to use different implementations. Since this is the case, we are in this study also interested in attempting to extract which programming languages applications using GraphQL use and which GraphQL implementations they use.

GraphQL was initially developed as an internal project at Facebook in 2012. It was released as an open-source project in July 2015, still under Facebook's management. Then finally, in November of 2018, the project moved from Facebook to the just established GraphQL Foundation. We will, as such, focus on data mining data from 2015 to the present year, 2022. As of early 2022, no up-to-date historical studies of GraphQL history exist. However, one notable study conducted in 2019 can give us some insight into GraphQL's history, at least until that time. The study analyzed public GitHub repositories to create timelines for how GraphQL had been adopted since its release up until mid-2018 [10]. We will compare and discuss their results later in this paper.

The research question we will explore are the following:

- **RQ1**: What insights are achievable with our method for creating a historical timeline, using GraphQL as an example.
- **RQ2**: What insight are we able to gain of a software subject by performing a high-level analysis of its programing language metadata.

BACKGROUND

We use GrapQL as our test keyword due to a suggestion by Olaf Hartig[2]. Hartig, a researcher at Linköping's University, was a stakeholder in the study [10] above and wished to see a more recent version of some aspects of that study. He also assisted in acquiring access to said study's research materials, which we used as an inspiration for how to gather and analyze GitHub and Libraries.io data.

[1] https://graphql.org/
[2] http://olafhartig.de/

THEORY

Data mining GitHub

With over 40+ million public repositories [4], GitHub provides a large dataset to study. GitHub offers several API endpoints that external programs can use to interact with their database. These API endpoints include a REST API. GitHub's APIs are restricted and enforce several limitations on connecting clients called *rate limitations*. Different resources have different rate limits. For instance, the REST `core` resources have a rate limit of 5000 uses per hour, compared to the `search` resources that have 30 requests per minute as a limit [5]. The data a typical user can retrieve and inspect are those marked as public. This data includes public projects, users, and code repositories.

Alternative historical GitHub data sources

When discussing GitHub data, we must mention alternative historical data sources. Two such sources are *GHtorrent*[3] and *GH Archive* [4]. Both archives only store the metadata that GitHub generates. Not repository source code or its changes. If stakeholders desire to extract historical source code, they need to return to GitHub, hoping they still have that version saved. GHTorrent and GH Archive intends to preserve the public GitHub timeline for future use, with a slight difference in philosophy. The difference in philosophy is that the GH Archive wishes to store as much GitHub data as possible. While GHTorrent wishes to make their collected data easier to analyze for users. This difference is apparent in how they store their data. GH Archive stores GitHub events into an activity log. While GHTorrent collects data into a MongoDB database and later into MySQL dumps with monthly releases[14]. GH Archive has stored log files from 2012 to today, while GHtorrent only has archived data from October 2013 to June 2019.

GH Archive's colossal data may seem desirable to this study until we apply our current research goals and study scope. GH Archive is accessible through Google's BigQuery. BigQuery offers a free 1 TB of data processing per month. We assumed this to be insufficient since we wish to search through several years worth of data. Also, GH Archive's lack of data structures makes it impossible to calculate how much data we would need to process to accomplish our goals. Therefore, we do not incorporate its data into our study. GHTorrent may also seem appealing. However, research conducted in GHtorrent infancy noticed several glaring issues. Two noted issues important to this study were: That GitHub's API and its response structure may change. Such a change could create a problem where the new API's response structure requires reformatting to fit GHtorrent's stricter schema, potentially losing valuable data. Secondly, GHtorrent's archiver could miss some GitHub events. Such a miss could occur if some fault happened in the GHtorrent data mining software or something malfunctioned in its network connection to GitHub[7]. Also, as seen in GHTorrents database schemas, it does not track changes to a repository metadata. It mainly tracks changes to a repositories content. Owing to the above and that GHTorrent only has gathered data up to June 2019, we did not integrate it either.

[3]https://ghtorrent.org/

[4]https://www.gharchive.org/

Problems with using GitHub as a data source

Previous research has noted several potential pitfalls when studying public GitHub data. A paper[9] from 2014 identified nine perils. The ones relevant in our case are the following. *"A repository is not necessarily a project"*[9], which in other words is that an open-source project may employ several repositories for one project, that it is not always a 1:1 relationship. *"Most projects are inactive"*[9], most projects, and thus repositories, may not get continuous updates or even bug fixes. However, the inactivity may be due to different reasons. A repository may be declared complete, deprecated, or dropped due to lack of interest. *"Two thirds of projects ... are personal"*[9]. *Personal* repositories can exist for several reasons. Some repositories were created purely for personal development, while others are forks that never got merged back into their origin. Differentiating between personal and public repositories can be a challenge. We will discuss this later in the discussion.

A more recent study from 2020[1] into GitHub data mining highlights additional problems and reinforces some old ones. Relevant to this study are the following. *"Richness and variety [of data] might lead to seemingly contradictory results if the construct definitions and methodologies are not clearly delineated"*[1]. This study follows the available data structures retrievable from GitHub's REST API and analyses the data we collected into them without reorganization. Since we only analyze repository metadata, not its content, which is less open to interpretation, we are unlikely to have this problem happen. *"A project can no longer be viewed as a single repository, but a constellation of repositories"*[1]. Which we already discussed in connection to the prior study. *"Many FLOSS [Free and Open-Source Software] projects use multiple programming languages[,] and they should be carefully classified"*[1]. This fact is important to us, which we discuss later in the theory. *"Researchers should not assume that all sampled projects are software-based and must screen projects for appropriate type"*[1]. GitHub repositories may contain documentation, images, and other digital assets that may or may not be relevant to what we want to study. If we desire to get a pulse for how often a keyword appears, then it may be preferred to include them, while if we are only interested in the context of software code, we may wish to exclude it. This paper[1] also notes that researchers should consider what now has become named *community health score*[1]. One method of measuring a community's community health score is to evaluate whether they fulfill the *recommended community standards* [5], which is the method that GitHub uses. GitHub exposes this data to users through its API, where users can retrieve a list of which standards a project fulfills. The study[1] endorses combining GitHub's API data with other sources such as GHtorrent to create a more complete data set to study. And that *"GitHub search API, as well as GitHub topics, can be a versatile tool for classifying repos and studying trends on GitHub"*[1]. However, accomplishing this is above the scope of this study, and we will instead discuss it in the discussion.

[5]https://opensource.guide/

Data mining Libraries.io

*"Libraries.io gathers data from **32** package managers and **3** source code repositories. We [Libraries.io] track over **2.7m** unique open source packages, **33m** repositories and **235m** interdependencies between them"*[13]. Libraries.io makes this data available through their website[6] or via a publicly available REST API. In our preparation for this study, we could not find any equivalent studies to those conducted for GitHub. Regarding what problems researchers may encounter when mining and analyzing Libraries.io data. Consequently, we were more blind here than we had wished when beginning our study. Libraries.io occasionally publishes their complete dataset in the form of CSV files. This dataset is used in several studies[18, 16]. The latest released dataset is the one released in 2020[13], which was two years before this study, and therefore deemed insufficient for this study. We instead use Libraries.io REST API. Which, similarly to GitHub's API, is rate limited. With a universal 60 requests per minute limit[12].

Duplicate repositories

There exist several studies which attempt to find similarities in git repositories. These studies are categorized into two types, *high-level similarity* and *low-level similarity* analysis[2], in other words, repository metadata and repository content analysis. Two high level studies are *RepoPal*[17] and *CrossSim*[15]. RepoPal uses three heuristics to perform its deduction. These are, *"repositories whose readme files contain similar contents are likely to be similar with one another"*, *"repositories starred by users of similar interests are likely to be similar."*, and *"repositories starred together within a short period of time by the same user are likely to be similar"*[17]. CrossSim has a similar but more nuanced approach. The relevant data points they analyse are: `commits`, `hasSourceCode`, `develops`, `stars`. Both mentioned *high-level similarity* studies solely focused on accuracy, not time complexity nor repository size universality.

METHOD

The source code for this project is available at GitLab.LiU.se[7].

Method strategy

Our program aims to download and analyze metadata from both GitHub repositories and Libraries.io libraries. We do this by interacting with their respective metadata APIs and saving the collected data to a relational database. We must also carefully decide what metadata to request since all APIs have rate limitations.

Shrinking history

Before describing our method, we must introduce one additional aspect that reduces the validity of our current study and any subsequent studies. GitHub's and Libraries.io's historical data is ever-changing and lossy. GitHub takes a great effort in saving repositories' content history and Libraries.io in storing a library's versioning. However, they are not as careful in archiving their repository or library metadata.For GitHub, this would be, renaming a repository, changing its description, or removing it. Correspondingly with a Libraries.io library. If some metadata has changed since its initial creation, we cannot find out what it originally was, at least from the services themselves. In other words, someone could add our keyword anytime between the data's creation to its latest version. The main reason for this problem is that neither service employs metadata version control. The only indicator GitHub has that reveals if a repository's metadata has changed is the `updated_at` field. According to a software engineer at GitHub, `updated_at` is an indicator for changes such as changing the name, rewriting the description, or changing the repository's primary language. This engineer also notes that a repository's primary language will automatically change if someone added a significant amount of new code in a new language to it[19]. This last fact is especially troublesome to us since it can occur without any conscious intent on the repository owner's part, considerably limiting our possibility to rely on the `updated_at` field as an indicator for metadata validity. We could adjust for this by incorporating one of the previously mentioned external archives, GHtorrent or GH Archive. But only if we can deal with the problems they accompany. Libraries.io has, to our knowledge, no external archives, which limits our options. The only workaround we could envision would be to use the Libraries.io data releases, which would serve as Libraries.io database snapshots. We could use these to extract the difference between them and create a database timeline with up to years as the lowest granularity. All the problems discussed in this section present a fundamental issue. Our results are, in principle, unreplicable. Since the historical data we analyze can, and most likely will, change.

Study environment

This study had a combined research time of 10 weeks which included performing research, writing this paper, writing the data mining program, and running said program. This study had a low-end Dell PowerEdge R330 running Ubuntu 20.04 LTS server edition, which we used to run all the Python3 data mining scripts. This hardware is not crucial to this study's replicability, and any computer with sufficient disk storage can suffice. The data-mining program(s), henceforth also called *scraper*(s), requires a MySQL database to save its results. The database is initialized with the provided database table schemas found in `GH.sql` and `LI.sql`, summarised in figure 1. The scraper use Python3.8 with some additional libraries, listed in `requirements.txt` found in our repository.

Using GitHub

The GitHub scraper consists of four data mining algorithms, where we data-mine metadata such as repositories, owners, and commits. We primarily mine for all repositories containing our keyword then we gather associated data. The left part of figure 1 shows the tables to which the GitHub scraper saves data. The scraper interacts with GitHub through the URL `https://api.github.com`. Henceforth, when we reference a GitHub API endpoint, we will write only the resource path where we imply that this protocol and domain name precedes it. For example, `/search/repositories` translates to `https://api.github.com/search/repositories`. Most GitHub

[6] https://libraries.io/

[7] https://gitlab.liu.se/gusbo010/github-open-source-scraper

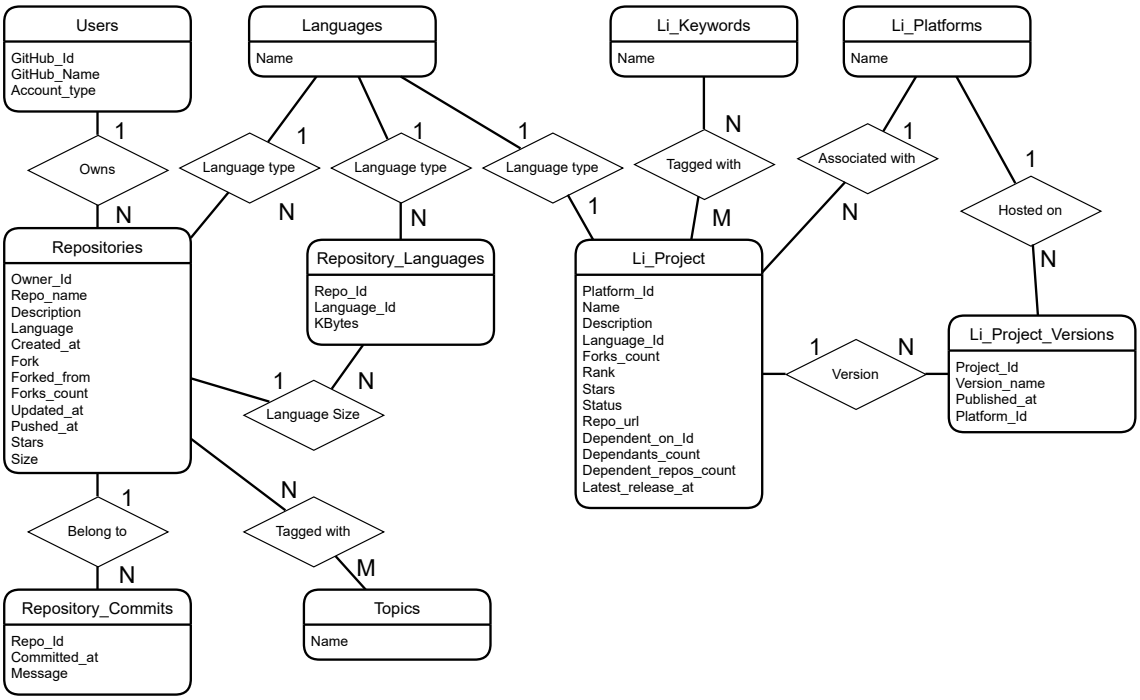

Figure 1. Database schema

API endpoints can take arguments to specify and modify the resources fetched. In the cases where it is possible to retrieve several results per API request, we always include the argument `per_page` which we set to the max of 100. And in cases where we cannot receive all results in one go, we include the argument `page`, which makes it possible to iterate over the available response parts called pages. The page range allowed is one through ten, resulting in a maximum of 1000 results retrievable per endpoint request specification.

GitHub Scraper

The GitHub Scraper is divisible into six mining actions. These are: `scrape repos`, `scrape in topics`, `scrape in readme`, `scrape fork cascade repos`, `scrape languages`, and `scrape search term in commits`. The first three scraping actions are similar and deal with finding repositories containing a keyword through GitHub's search API, in our test case with the keyword GraphQL. The API endpoint used here is `/search/repositories`. This endpoint allows for several arguments. We use `q` for specifying a query and `sort` to sort the results. The `q` query can take several arguments, such as a keyword, fields to search, and timespan to search. The search endpoint also includes an option for whether the result should contain forks. We set this to true, which we will discuss why later in the method.

The explanation why three mining actions are dealing with the same endpoint, instead of two or simply one, is due to the first problem we encountered in developing our mining scripts. The query string `q` takes a `in` qualifier that specifies what fields GitHub search should look in. The possible options are `name`, `description`, `readme`, and `owner/name`. The API documentation states that *"When you omit this [`in`] qualifier, only the repository name[,] and description are searched"*[6]. We quickly realized this to be incorrect, or at least inconsistent, as of April 2022. A request of `/search/repositories?q=graphql` compared to `/repositories?q=graphql+in:name,description` resulted in a difference of over a thousand results. These two requests should give the same result, according to the documentation, but evidently did not. We did not have time to investigate this in any depth. Our best guess is that when omitting the `in` qualifier, GitHub searches additional fields, likely topics. We have filed an issue and pull request for this, which as of writing this paper, has not been resolved[8]. Due to this, we decided to split the search action into three parts. One where we use `in:name,description`, another with `in:readme`, and a third with `topic:graphql`. Searching with `topic:graphql` is essentially the same as `in:topics`, except GitHub does not allow topics as an in qualifier.

Search API problems

Our second problem with GitHub was also related to the search API, specifically when sorting results. The sort options available are `stars`, `forks`, `help-wanted-issues`, `updated`. None of which sorts result in a consistent order, as they are all subject to change. While this may not be relevant in small or specific searches, it creates an issue for us in our attempt at mining repositories in the millions. If our search results in more data than can fit on one page, we must fetch it over several pages, creating a risk of items getting reordered in-between page retrievals, causing us to miss some items. Instead of estimating some expected data loss, we used another

approach. Thanks to the GitHub option to specify a time range in our request, we can split a large request into several smaller time range limited requests. To attempt to receive below 100 results per search. The granularity in the time range option is down to one day, predominantly allowing us to hit our target of 100 or below. We still anticipated some cases where this would not be possible, where we receive more than 100 results from a day and therefore included a fallback strategy of page iteration. We also decided to sort by forks, assuming it to be the characteristic that changes least often.

GitHub algorithms

The final result of what we ask in the search API thus looks like: `/search/repositories?q=graphql +fork:true +created:from_date..to_date &per_page=100 &sort=forks &page=1`. The algorithms we use to run all three mining actions can be summed up as follows in listing 1.

Listing 1. Search scraper algorithm

```
curr_from = start_date
curr_to = end_date

while True:
    repos, total_count = github_api_action(curr_from,
         curr_to)
    if total_count > 100:
        if curr_to == curr_from:
            save_action(repos)
            pages = ceil(total_count/100) if
                 ceil(total_count/100) <= 10 else 10
            for page in range(2, pages+1):
                repos, total_count =
                     github_api_action(curr_from, curr_to,
                     page)
                save_action(repos)
            if math.ceil(total_count/100) > 10:
                lost_data_count = total_count - 1000
                log(f"Lost {lost_data_count}")
            curr_from, curr_to =
                 _date_interval_window_algorithm(curr_from,
                 curr_to)
        else:
            curr_from, curr_to =
                 _date_interval_window_algorithm(curr_from,
                 curr_to, exceeded_window = True)
    else:
        save_action(repos)
        curr_from, curr_to =
             _date_interval_window_algorithm(curr_from,
             curr_to)
    if end_date < curr_from:
        break
```

The second mining algorithm we developed was a cascading fork miner, which searches through and downloads each repository's registered forks. Due to time constraints and that we included repository forks in our search algorithm, our final dataset did not utilize data gathered using it, and we consequently omitted it from this paper.

The third and fourth mining algorithms, data mining actions five and six, deal with gathering data points from each repository. These data points are `repository_languages`, and `commits_with_keyword`. The repository language data is accessible through the `/repos/user_name/repo_name/languages` endpoint. And we search commits using `/search/commits?q=repo=user_name/repo_name +graphql &sort=committer-date &order=asc &per_page=100 &page=1`, which is very similar to how we do our initial search, except per repository instead. We gather language data to use later to answer RQ2, which is the fifth mining action. This language data contains the name of the language and its size, in kilobytes, in the repository. The sixth overall mining action is to use the search API again to search for commits in each repository for commits containing our keyword. We do this to create a second approach to answer RQ1, which we discussed in the method. The last two mining algorithms can be abstracted to listing 2.

Listing 2. code.python3

```
repo = get_repo_with_data_not_scraped()
while repo != None:
    data = api_action(repo.todo)
    for item in data:
        save_repository_data(repo, item)
    mdbc.set_data_scraped(repo)
    repo = get_repo_with_data_not_scraped()
```

Libraries.io Scraper

From Libraries.io, we similarly intended to data-mine metadata. In this case, libraries. A library's metadata includes, among other things, a name, a description, and a list of its released versions. We list all collected data points in the right part of figure 1. Data mining Libraries.io is a whole other experience than GitHub. Firstly, Libraries.io has around 20 API endpoints available, while GitHub offers over 100 endpoints. Secondly, we received several `HTTP: internal server error`s from their APIs. The reasons for these errors were by our experience ranging from requesting too much data to straight-up unknown reasons. We initially envisioned doing something akin to what we have done for GitHub. But after receiving thousands of server errors in our attempts, we limited our scope to a top-level search for our keyword.

The search we perform interacts with the API endpoint https://libraries.io/api/search, in which we also encountered problems. The Libraries.io search API, similar to GitHub's, can be provided with several arguments. These include `q` for a search term, `sort`, `order`, `per_page`, and `page`. Here we can provide a query, sort it, and iterate over available results divided into pages of 100 items. Thankfully, Libraries.io has a consistent sort option, which is `created_at`. The problem we encountered here was that after iterating up to page 101, we always received an `HTTP: internal server error`. An issue has been filed to Libraries.io for this since November 2021 and has yet not been resolved[11]. As such, we must add a layer to our data mining algorithm, which we do by narrowing our search by specifying a specific language. Libraries.io search API allows us to accomplish this by specifying a language with the argument `language`. With all of this explained, an API request to the search endpoint will look like https://libraries.io/api/search?q=graphql&sort=created_

at&order=asc&per_page=100&page={page}&languages={language}
And our data mining algorithm for Libraries.io becomes as is shown in listing 3.

Listing 3. code.python3

```
for language in libraries_io_languages:
    data = api_action(page = 100, language)
    if len(data) >= 100:
        log(f"Unable to mine data for {language}")
        break
    for page in range(1,101):
        data = api_action(page, language)
        if len(data) == 0:
            break
        for project in data:
            save_project(project)
```

Matching project to implementation language

To attempt to answer RQ2, we can use two different approaches. The first and likely most correct approach would be to use *Libraries.io's*. With the help of Libraries.io, we can find open-source projects related to GraphQL and their dependencies on other projects. By using Libraries.io dependency data, we can attempt to map up and calculate what libraries are popular. We can also try to estimate which implementation languages are prevalent currently. Libraries.io also keeps track of where the different projects host their code, which in some cases will be GitHub, allowing us to compare this approach to our second approach. Our second approach is to utilize GitHub's linguist library repository report. Through GitHub's API, it is possible to gain a report over the programming language used in a repository. But as a study[1] aptly points out, GitHub's language report is only based on bytes of code of a specific programing language, rather than any more advanced metric such as importance to the project[1]. Hence, we will use the naive assumption that the dominant programming language is a predictor of what GraphQL library implementation it uses.

Extracting patterns in the data

After filling our database, we need to decide how to analyze it. We are sure there is an abundance of possible patterns you could extrapolate from the dataset. However, we are interested in historical changes, that is, patterns over time. GitHub and Library.io both have timestamps in several data types. We have collected GitHub repository `created_at`, `updated_at`, `pushed_at`, and commits `committed_at`. Libraries.io has `latest_release_at` and library `published_at`. We use these below in the results to plot several figures shown in our result. We have noted additional possible plots we did not include in this paper. For instance, detecting when the most stared or forked repositories were created, what programing languages usually accompany GraphQL applications, which topics accompany GraphQL, to something complex such as what language implementation was popular at specific times.

Related study difference

The study[10] we take inspiration from constructed historical timelines as total projects created at specific dates. They also drew a plot where they tracked when commits containing *"GraphQL"* were committed to the repository. They have not published their source code, but by our rough estimation, 70% of our program and their program behave the same. They used Jupyter Notebooks[8] to write Python3 and Javascript code, while we opted for pure Python3 scripts. Another difference is that they store their data in CSV files while we use a MySQL database. The justification for our shift to pure Python3 was our familiarity with it, that we wanted to rewrite their program to detect potential mistakes, and that we had an external server we wanted to use. Our shift to a MySQL database was due to our desire to have more accessible data that could include more complex relationships. The 30% difference in behavior comes from correcting some of their data collection methods, collecting more data from GitHub and Libraries.io, and adding functionality to deal with the more sizeable results we receive from later dates.

Duplicate repositories

Using one of the *High-level similarity* studies we mentioned in the theory section, we could eliminate some duplicate repositories. We left off that theory section by saying that neither study focuses on time complexity, only accuracy, which is a problem for us. We plan to collect a several magnitudes larger dataset than what they used in their analysis, their hundreds compared to our million. Our program would also be required to scrape additional data fields they used in their method, which would extend the scraper's runtime, which we discuss next section. Both study methods, by our analysis, have a time complexity of O(N!), which would require a runtime we don't have available. Our program also collects repositories of all sizes, which may undermine any value their methods give since they only used a selection of repositories in their tests. Due to the above, we could not integrate either study's method.

Time complexity

Lastly, we need to discuss the time complexities of executing our scraper. As mentioned before, GitHub has several rate limits. However, these rate limitations are not the same as *item retrieval* rates. The `core` rate limit of 5000 per hour links to when you access specific resources, in our case, a repository's language data and forks. That is a flat 120k request per day. However, if we regard how many results we can retrieve per request, we get an item retrieval limit of up to 12M items per day, 100 times as much. We fall into the lower retrieval rate if we mine for language data. And in the case of mining repository forks, while unlikely, we approach the 12M item retrieval limit.

The search API rate limit has a similar situation. The search rate limit links to us performing a search action. The base rate of 30 requests per minute is equivalent to about 43k requests per day, which in the cases where we can retrieve 100 results per request, sums up to 4.3M items per day. Here we come to one of the cruxes of why we included forks in our first three data-mining actions, where we search the whole of GitHub for our keyword. In our development, when we used the cascading fork algorithm, we noticed that most repositories had well below 100 forks, resulting in our effective item retrieval rate

[8] https://jupyter.org/

to close in on the lower bound of 120k items per day of the `core` rate. However, when we used the first mining algorithm, we usually neared the higher bound rate of the `search` rate of 4.3M items per day. We consequently opted to find the forks when we did our initial search instead of later retrieving them through the cascading fork scraper algorithm. The balancing act here is that if we include repository forks in our search algorithm, we can find forks that have become related to our keyword after their split from their parents, which we would miss in our cascading fork algorithm. And in the case where we only run the cascading algorithm, we could find forks that had removed their identifiers of being related to our keyword, with the risk of finding repository forks created before the parent began using the keyword.

If the GitHub scraper finds around 1M results over its three first scraping actions, we can expect an initial runtime of 6+ to 17+ hours. The explanation for this speed is that they can approach the 4.3M item per day retrieval limit. The range in runtime is mainly due to the duplication of effort we discussed in the theory. These 1M results need to have their languages mined, which takes 8+ days due to the lower bound of the `core` item retrieval limit. Then the scraper also needs to search through each repository's commits for our keyword, which takes 24 days, resulting in a total runtime of around one month, not including the cascade fork algorithm, if run.

Libraries.io has a rate limit of 60 requests per minute which is 86k per day. Similarly to GitHub, it is possible to retrieve up to 100 results per request depending on the API endpoint, which gives us an upper bound of 8,6M item retrievals per day. Luckily, the search endpoint is one of those. Performing a search for our keyword on the Libraries.io website shows us that there are around 14k results to fetch, translating to a theoretical 4 hours runtime. The actual runtime is double that, due to various reasons, such as slow request responses and internal server errors.

RESULT

After data-mining all of our desired data, we plot our results into the folowing figures. As can be seen in figure 1, we track in what GitHub field we found our keyword in. The available fields, or tiers, are `in:name`, `in:description`, `in:topics`, and `in:readme`. This is relevant as we can plot this data into different lines, shown in figure 2, 3, and 4. Note that one repository can have the keyword in several fields, thus be counted in several lines.

Figure 2 shows repositories created each month that includes our keyword in some fields, with and without forks.

Figure 3 shows how many repositories were active in a month. We defined activity here as the repository's creation day to the latest of either `pushed_at` or `updated_at`. In other words, the GitHub repository was created that month (or before) and has had at least one change made that month (or sometimes) in the future, which is not the same as the repository having had some actual activity that month.

Figure 4 shows what month we first found a commit mentioning our keyword, the first commit containing our keyword once per repository, sorted into months.

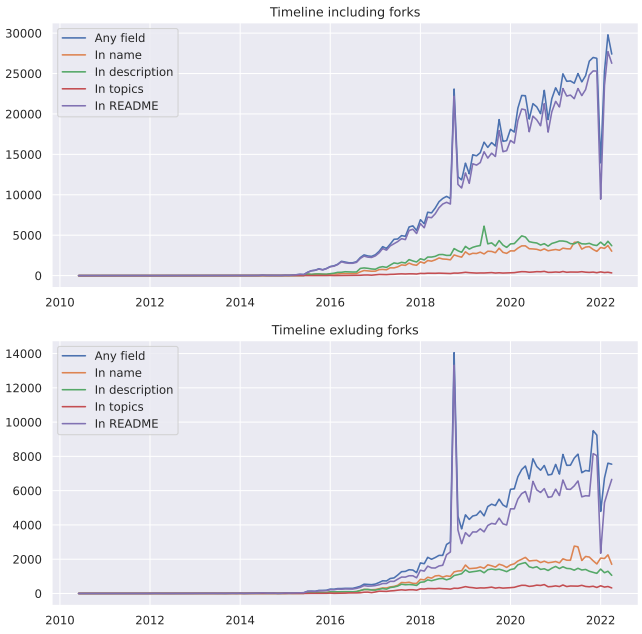

Figure 2. Repositories created by month

Figure 3. Repositories 'active' by month

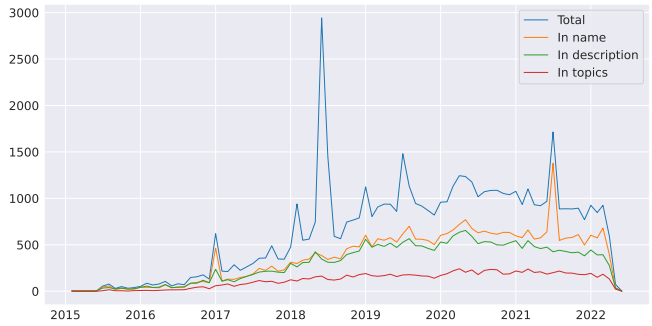

Figure 4. First occurrence of keyword in repository commits

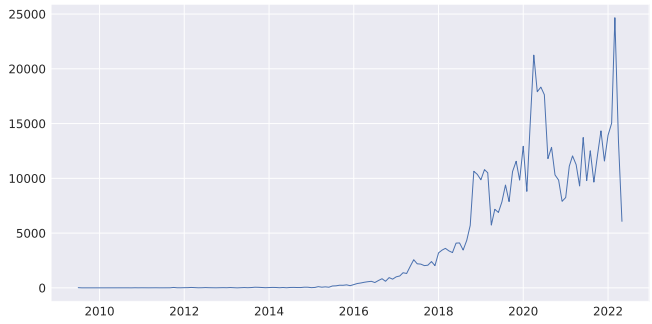

Figure 5. Library releases grouped by month

Language	Tot	Name	Desc	Topc	Readm
NONE	50.27	28.50	36.82	50.27	52.67
JavaScript	26.43	39.31	31.53	26.43	24.79
TypeScript	8.33	12.56	13.94	8.33	7.78
CSS	2.57	0.66	0.78	2.57	2.68
Python	2.21	2.12	2.50	2.21	2.22
Go	1.64	1.96	1.95	1.64	1.66
Java	1.45	3.66	2.36	1.45	1.35
PHP	0.86	2.43	2.02	0.86	0.78
Ruby	0.85	2.18	1.54	0.85	0.76
HTML	0.82	0.68	0.57	0.82	0.80
C#	0.68	1.75	1.25	0.68	0.62
Swift	0.49	0.24	0.41	0.49	0.51
Vue	0.47	0.48	0.51	0.47	0.43
Dart	0.41	0.40	0.30	0.41	0.42
Other	2.53	3.07	3.54	2.53	2.53

Table 1. Language distribution in % by field keyword was found in

Language	Percent
TypeScript	39.04
JavaScript	27.98
Go	6.48
PHP	5.07
Java	4.86
Kotlin	3.33
Python	2.60
C#	1.99
Rust	1.49
Scala	1.47
Ruby	1.45
Clojure	0.99
Dart	0.61
Elixir	0.50
Other	2.14

Table 2. Language distribution in Libraries.io keyword search

Table 1 shows a breakdown of which languages the repositories declared as their primary language, sorted into what fields we detected the keyword. For instance, 2.5% that had our keyword in the description had Python as their primary language. *NONE* represent the case where the repository lacked language association.

Figure 5 shows the number of library releases released by month. The data presented in this figure is a superset of Libraries.io library's creation date. Since, in Libraries.io, the creation date of a repository is the same as the date of its first release.

Table 2 shows what languages a library declared as its primary. Since we only found 14k libraries, we deemed it extraneous to split it into what field we found our keyword. And as therefore only presented the total.

Database

Lastly, we wanted to mention how much data we lost in the different steps. The GitHub scraper, even with the help of page iteration as a backup strategy, lost close to 3k repositories. Between the GitHub scraper's first three mining actions and the last two, we lost around 50K repositories. The Libraries.io scraper missed 100 results, 75 of which were later possible to retrieve manually. And on a side note, in our attempt at mining the dependants on found libraries, we could not download dependants from 2k libraries. Most of which we found to be libraries with numerous dependants. Our final database size landed around 20 GB, though we believe half of that size comes from the MySQL logs generated during our development.

DISCUSSION

Result discussion

Using our data and naive heuristics, we can see that GraphQL has had new adopters each year since its release in 2015. From GitHub, we can see that GraphQL has new repositories linearly added to GitHub and an exponential increase in their overall activity. We can see that repositories mentioning GraphQL in their README have increased more than in any other field, likely due to the faster adoption from secondary and tertiary users of GraphQL through other software projects. We also see some months that noticeably deviate from the typical course, which we did not have time to investigate. These deviations are likely related to significant events in GraphQL or GraphQL repositories. We can see a similar story in the libraries' data gathered from Libraries.io but with even more discrepant months. We did not have time to data-mine how many software projects use said libraries, but if we had, we suspect we would see a more explosive growth than our GitHub activity data show. We can from GitHub and Libraries.io see that web-focused programming languages are

the most used, whereas the other languages are lower due to them being for backend use.

Related study comparison

Comparing our results to the study[10] we mentioned in the introduction shows that our results align with theirs. That includes figure 2 about the repository created per month and figure 3 for the first occurrence of a keyword in a commit. Though as we suspected, we found more results than they did. In the case of creation date, we likely found additional data due to what we discussed in the method about repositories beginning to use our keyword after its initial creation. Then also, in the first occurrence of our keyword in commits, the additional results we found could be caused by us gathering data in a slightly different way.

Threats to validity

The rationale behind saving which fields we encountered our keyword was to gain insight into at what level GraphQL is relevant to that GitHub repository. In our reasoning, using GraphQL as a repository's topic is the clearest indicator that it is related to our keyword. However, GitHub introduced repository topics in early 2017. All repositories created before 2017 must have had the keyword added after its creation. Likewise, some repositories created after 2017 will likely have had the topic adder later than its creation date. After topics, we reason that a repository using the keyword in its name is the second-best indicator that it is related. Our rationale here is that the GitHub name field has a character limit of 100, thus forcing owners to write a name that only includes the most relevant phrases and keywords. The description has no such limitation. The consensus is to keep a repository description shorter than 256 characters. The readme is likewise unrestricted in size. A readme is typically more extensive than the description and thus has more chance to mention our keyword, which is why we encountered it the most in our search.

Figure 2 contains results that predate GraphQL's public launch, which does not only include repositories that used GraphQLs before its public release but also repositories that later added GraphQLs in some fields. The cause is that GitHub's metadata is not retroactively consistent, which we discussed in the method. A project created in 2016 could begin to use the keyword 2020, and we would be none the wiser, at least with our approach.

A solution to figure 3's activity resolution problem would be to data-mine each repository commits. But as we have mentioned numerous times would increase the scraper's runtime. Another weakness with 3 and 4, and the reason that they seemingly drop off at the beginning of 2022, is that many projects only have sparse commits. And as such have not had any activity this year, even though they may still be in development.

A problem with all our figures is that they do not compensate for the importance of each data point. We assume every item to be of equal importance. We found no strategy in our investigation that can handle this, at least not for the variety of repositories and libraries we collect. This problem also exists in both of our language tables. After collecting all language data, we can say that even our most reasonable assumption in section the theory is still naive. The approach we will recommend for future attempts at extracting language data from Libraries.io is manually picking relevant libraries and data-mine their dependent libraries. Doing this would lead to a better result in the case where we want to find what implementations they use, not what language the libraries themself use. Also, due to the additional Libraries.io scraper layer, we circumvented any libraries that did not have any language associated with them, missing them from our language breakdown.

To add to what we discussed in the method about the shrinking GitHub and Libraries.io history. The consequence of this shrinkage is that any study completed before this one will have had access to more correct data than we currently have. Because the closer the data was collected to its creation, the less time has passed for it to change in any way we cannot trace. However, when collecting more recent data, other problems appear. One of these problems is with what we call *temporary data*. *Temporary data,* is data that users create and remove within a limited time. Examples of such data would be, testing repositories and pull request repositories.

Deciding what repos to count in historical graph

A third problem our study has is with deciding what repositories to include. We include everything from minor code repositories, forks, and personal code repositories, to actual duplicates. One solution we mention is to use *community health score* [1]. Some factors used when calculating *community health score* are: `total_commits`, `max_days_without_commits`, `max_contributions_by_developer`, and `closed_issues` [3]. With the help of the health score, we could more efficiently pick out which results we should analyze. Yet, here again, we are constrained in study and runtime. Downloading all committed data for each report and developer contribution data would add a substantial time to our scraper runtime.

Lastly, we want to discuss two issues we noticed after we finalized our database schema. The first issue we detected was that one commit could belong to several repositories. That two repositories had the same commit saved twice to the database should not affect our results, only reduce our database storage efficiency. The second problem was that a library could depend on several libraries, which is apparent in hindsight. That we could not set several repository dependants is likely one of the many things that hindered our attempts at data mining library dependants.

Future work

One facet possible to study further to speed up GitHub data mining would be to examine how to best use the various rate limits that GitHub has. For instance, we have not mentioned GitHub's GraphQL API, which has its rate limits. With the help of the GraphQL API, it could be possible to retrieve some data points more efficiently than through the standard REST API endpoints. Our linear data miner algorithm is also woefully inefficient. Creating dependency data for a multithreaded scraper could speed up the execution time. Especially if the

scraper had a more intelligent way to mine forks, where if the parent had been data mined, the children would not be data mined. Also, as we mentioned in the theory, there are alternative sources for GitHub metadata. Similar to the GraphQL API, we could use them to speed up data retrieval.

We would also like to mention some additional problems future studies could expand on. There are several data types that GitHub offers that we did not include. For instance, discussions, issues, and commits. All of which are useful, among other things, to estimate how active a repository was. Also, if researchers could conclude which sorting option in GitHub's search API was most consistent, it would alleviate some we encountered gathering repositories. Or simply if GitHub decides to implement such an option in the future. Then with Libraries.io, if researchers found a better way to collect libraries and their dependents, it would be easier to create dependencies graphs. Or if the Libraries.io development team resolved the 100-page search limit. We believe Libraries.io has much more potential than was realized in this study. Mainly since it tracks libraries, which GitHub does not at all.

The facet we would be most interested in if a future study could tackle would be finding out at what rate GitHub, or git repositories in general, degrade in its history. Both in how fast metadata gets overwritten and how quickly a repositories content history stabilizes. We would call this *GitHub history volatility*. Such a study could involve a longitudinal study of a specific subset of GitHub data. To deduce at what rate different data points become historically stable, including the risk for different data types to be changed or removed. Such a study would help future studies attempt to extract historical data and evaluate the validity of their results.

Publishing our dataset

Neither GitHub nor Libraries.io users have, to our knowledge, any user agreement clause that says that unspecified third parties are allowed to archive their data. We find it ethically permitted to collect their public data for local analysis since the users of those sites are aware that they are publicly visible. However, we will not take any stance on storing such data for extended periods. Consequently, we will not publish our dataset. The main reason is to avoid conflicts with handling potentially private data. Any interested reader is free to use our methods or actual scripts to create a comparable dataset.

CONCLUSION

Despite the numerous problems and caveats we encountered in developing our data miner and the struggles from both data sources, we still believe that data mining GitHub and Libraries.io for historical data is worthwhile. We were able to use our method to gain some insights, such as:

- How the adoption of a software subject has been at different times.
- How active a subject was in public software.
- What programming languages were most associated with our subject.

The language association we found is at the least able to tell which programming languages usually accompany our keyword. Or, in the best case, show what GraphQL implementations are popular. Our approach is also cost-effective since it requires no payment, courtesy of GitHub and Libraries.io having their data freely available. The only tangible cost our method has is in requiring time to execute. But as we noted, have room to be considerably reduced.

REFERENCES

[1] Mohammad AlMarzouq, Abdullatif AlZaidan, and Jehad AlDallal. 2020. Mining GitHub for research and education: challenges and opportunities. *International Journal of Web Information Systems* (2020).

[2] Ning Chen, Steven CH Hoi, Shaohua Li, and Xiaokui Xiao. 2015. SimApp: A framework for detecting similar mobile applications by online kernel learning. In *Proceedings of the eighth ACM international conference on web search and data mining*. 305–314.

[3] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology* 122 (2020), 106274.

[4] GitHub. 2022a. GitHub total repositories search. (2022). `https://github.com/search?q=is:public`, last accessed on 2022-02-27.

[5] GitHub. 2022b. Resources in the REST API. (2022). `https://docs.github.com/en/rest/overview/resources-in-the-rest-api`, last accessed on 2022-05-27.

[6] GitHub. 2022c. Searching for repositories. (2022). `https://docs.github.com/en/search-github/searching-on-github/searching-for-repositories`, last accessed on 2022-04-26.

[7] Georgios Gousios. 2013. The GHTorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 233–236.

[8] gusbo010. 2022. Update searching-for-repositories.md. (2022). `https://github.com/github/docs/pull/17389`, last accessed on 2022-05-27.

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.

[10] Yun Wan Kim, Mariano P Consens, and Olaf Hartig. 2019. An Empirical Analysis of GraphQL API Schemas in Open Code Repositories and Package Registries.. In *AMW*.

[11] kirq4e. 2021. API request to Project Search with page > 100 returns Internal Server Error. (2021). `https://github.com/librariesio/libraries.io/issues/2880`, last accessed on 2022-04-26.

[12] Libraries.io. 2022. API Docs. (2022). https://libraries.io/api, last accessed on 2022-05-27.

[13] libraries.io. 2022. Libraries.io/data. (2022). https://libraries.io/data/, last accessed on 2022-04-27.

[14] Thaıs Mombach and Marco Tulio Valente. 2018. GitHub REST API vs GHTorrent vs GitHub Archive: A comparative study. (2018).

[15] Phuong T Nguyen, Juri Di Rocco, Riccardo Rubei, and Davide Di Ruscio. 2020. An automated approach to assess the similarity of GitHub repositories. *Software Quality Journal* 28, 2 (2020), 595–631.

[16] Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. 2021. Identifying Versions of Libraries used in Stack Overflow Code Snippets. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 341–345.

[17] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. 2017. Detecting similar repositories on GitHub. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–23.

[18] Chengming Zou and Zhenfeng Fan. 2022. GELibRec: Third-Party Libraries Recommendation Using Graph Neural Network. In *International Conference on Database Systems for Advanced Applications*. Springer, 332–340.

[19] Ivan Zuzak. 2015. GitHub API V3 : what is the difference between pushed_at and updated_at? (2015). https://stackoverflow.com/questions/15918588/github-api-v3-what-is-the-difference-between-pushed-at-and- last accessed on 2022-04-27.