The GHTorent dataset and tool suite

Georgios Gousios
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
Email: g.gousios@tudelft.nl

***Abstract*—During the last few years, GitHub has emerged as a popular project hosting, mirroring and collaboration platform. GitHub provides an extensive REST API, which enables researchers to retrieve high-quality, interconnected data. The GHTorent project has been collecting data for all public projects available on Github for more than a year. In this paper, we present the dataset details and construction process and outline the challenges and research opportunities emerging from it.**

***Index Terms*—dataset, repository, GitHub**

I. Introduction

During the recent years, Github has become the repository hosting site of choice for many Open Source Software (OSS) projects. Interestingly, Github provides a REST API to its full data set, making it an attractive research target. The GHTorent project uses the Github API to collect raw data and extract, archive and share queriable metadata. The created datasets have already been exploited in other work (analysis of the pull development model [1], analysis of drive-by commits and analysis of test incentives on social sites), while collaborations with external research groups are under way.

An outline of the project and bits of the implementation were presented in [2]. Since this work, we extended the collection process to an additional 15 API end points, stabilized the data and metadata schema and developed a service to collaboratively collect and share data. More than 900GB of raw data and 10GB of metadata have been collected and are available for download. In this paper, we present the finalized schema, go through the challenges and limitations of working with the dataset and outline research opportunities that emerge from it.

II. Data Collection

The primary challenge for the data collection process is the Github imposed 5,000 requests per hour limit for authenticated requests, while the event generation rate is already higher;[1] given that a single event can lead to several (even thousands) of dependent requests, it is not practical to assume that a single Github account will suffice to mirror the whole dataset. For this reason, GHTorent was designed from the ground up to use caching excessively (to avoid duplicate requests) and also to be distributed (to enable multiple users to retrieve data in parallel). We briefly describe how GHTorent employs those two mechanisms in the following paragraphs.

[1]On an average day, Github produces 200,000 events, or about 8,300 events per hour

The Github API supports two types of queries:

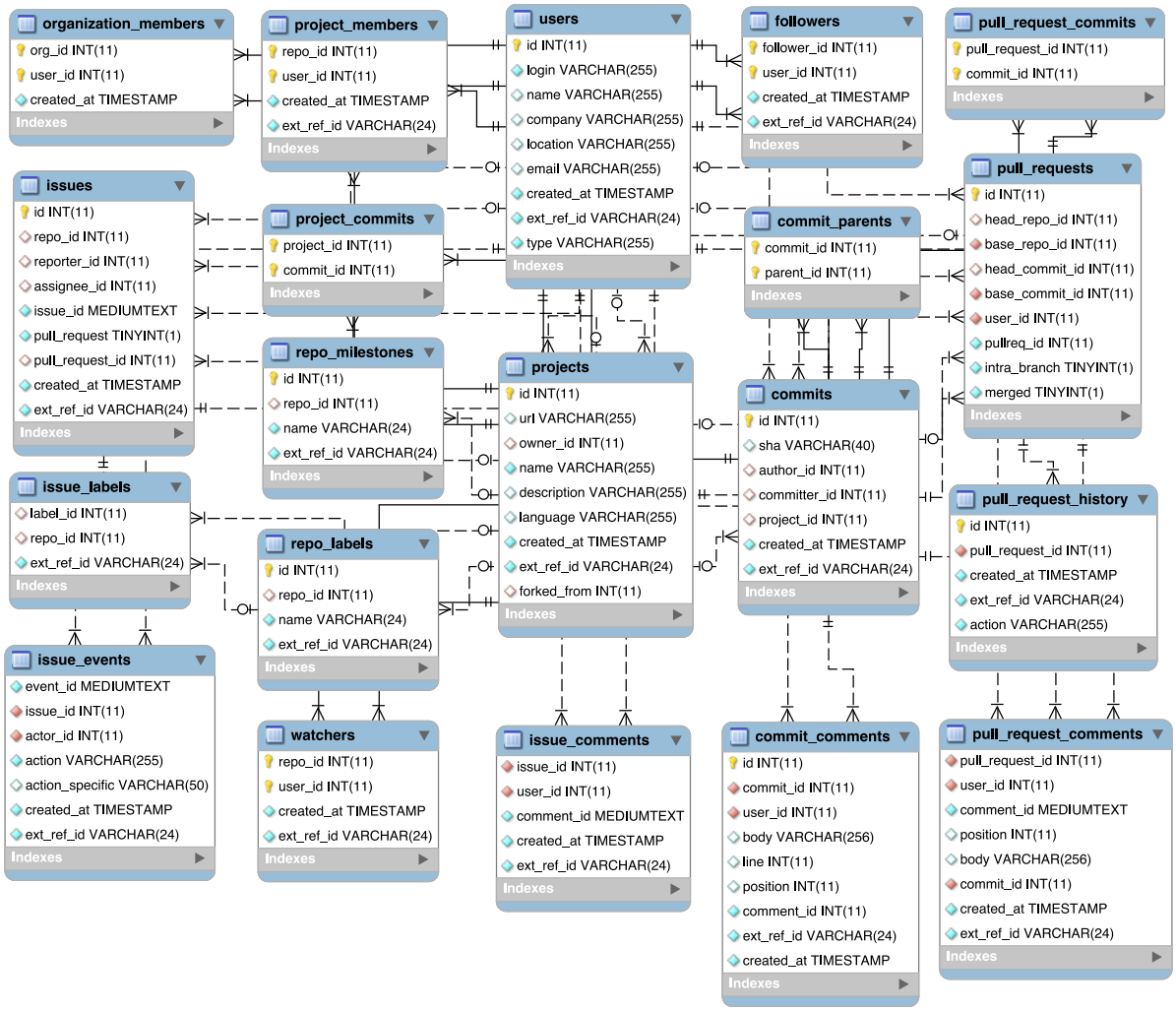
- Resource queries retrieve a specific instance of a resource. Per REST architecture mandates, the URL identifying a static resource remains constant after the resource has been initialized.
- Range queries retrieve a list of resources, usually related to a given resource. For example, the query `/{user}/followers` retrieves the followers for a user, while the query `/{user}/{repo}/commits` retrieves the commits for a repository. Paging is used to limit the amount of data per response. Range queries do not necessarily return the full entity instance for each item, but they usually include a URL where the item may be retrieved. As a result, a range query might result to several resource queries.

Resource query results can be cached very efficiently, as by definition they never change. Range queries are trickier to cache, as their result might change as the project evolves (new commits, new followers, etc); fortunately, by default Github serves newer results first, so it is enough to go through the first few pages of results only in order to retrieve the updated data. To cache the results per entity, we use a MongoDB database, which offers the added benefit of enabling querying on the raw data. Caching is also used at the HTTP request layer; GHTorent automatically serializes HTTP responses to disk. This avoids retrieving older pages in range queries twice.

The mirroring algorithm is based on a recursive dependency resolution process. For each retrievable item, we specify a set of dependencies, as they logically flow from the data schema (see Figure 1). For example, in order to retrieve a project, it is necessary to retrieve the owning user first; similarly, in order to retrieve a pull request, the project needs to be retrieved first. If any step of the dependency resolution fails, the whole item is marked as non-retrieved. The process was designed from start to be *idempotent*: every step of the dependency resolution may fail but once it succeeds, it will always return the same result. This design choice has been very important since it makes our data stores append only and the results of each step memoizable and therefore cachable.

Data schema

The data schema can be seen in Figure 1. Following Github's API organization, most entities belong to a project and contain entries corresponding to actions initiated by a user. The schema also records information about users,



Entity	Description	Raw data entity	Num Items
—	Events on repositories.	`events`	43,090,195
projects	Project repositories.	`repos`	1,326,900
users	Github users.	`users`	793,855
project_members	Users with commit access to the referenced project.	`repo_collabs`	983,629
organization_members	List of members in an organization.	`org_members`	34,924
commits	A list of all commits on Github. The `project_id` field refers to the first project this commit has been added to.	`commits`	29,978,291
project_commits	List of all commits to a project.	—	—
commit_parents	Commits that are parents to a commit.	—	—
commit_comments	Code review comments for a commit.	`commit_comments`	126,697
watchers	users that have starred (was watched) a project	`watchers`	7,744,619
followers	users that are following another user.	`followers`	1,797,343
issues	Issues that have been recorded for a project.	`issues`	2,326,069
issue_events	Chronologically ordered list of events on an issue.	`issue_events`	4,085,294
issue_comments	Discussion comments on an issue	`issue_comments`	2,886,006
pull_requests	List of pull requests for base_repo. Requests originate at head `head_repo/commit` and are created by `user_id`	`pull_requests`	1,144,251
pull_request_comments	Discussion comments on a pull_request	`pullreq_comnts`	2,228,894
pull_request_history	Chronologically ordered list of events on a pull_request	—	—

Fig. 1. Schema entities, their description, the corresponding raw data entities and the number of raw data items (Feb 15, 2013).

organizations and their members (organization_members), while commits are shared among projects, their forks, and pull_requests. Most entities are timestamped with their creation date (the `created_at` field), while for entities which can be in more than one states in time (e.g. pull_requests and issues), additional tables record those states in chronological order. For entities that correspond to API calls, the `ext_ref_id` field contains the unique identifier to the raw entity in the MongoDB database.

Workers for Data

The data collection was designed from the beginning as a decentralized process. Decentralization is mediated using the worker queue model; a message producer sends messages to the appropriate queue and several workers process messages, perform the requests and store the results in the shared database.

Decentralization enables collaborating researchers to contribute to the data collection effort, through what we call the *workers for data* model. In exchange for data collection *workers*, the project offers direct access to the live project *data*bases. Set up instructions for workers as well as a preconfigured virtual machine are offered through the project's web site. Apart from offering direct access to the data, the project also distributes dumps of the collected data, in both raw and relational formats. Depending on the use case, the relational dump might be enough for further processing. The dumps are distributed using the BitTorrent protocol. Furthermore, a query interface allows third party users to directly query an archived version of the relational database.

III. Challenges and Limitations

From a repository mining perspective, the GHTorent dataset has the following limitations:

Data is additive: Github is a dynamic site where developers, projects and wikis are created and deleted constantly. Despite the fact that the Github event stream reports additions of entities, it does not report deletions. This means that the information in the GHTorent database cannot be updated when a user or a repository has been marked as deleted.

Important entities are not timestamped: Github does not report timestamps for the watchers/stars and followers entities. This means that it is not possible to query the followers for a user or the watchers for a repository at a specific timestamp. As a workaround, GHTorent uses the timestamp of the event that is generated when a follow/watch action is performed, but this is only limited to the events that took place since the GHTorent project started collecting data.

Issues and pull requests: Issues and pull requests are dual on Github; for each opened pull request, an issue is opened automatically. Commits can also be attached to issues to convert them to pull requests (albeit with external tools). As a result, discussion comments for a pull request need to be retrieved from multiple sources, namely from pull_request_comments for code reviews and from issue_comments for pull requests. Moreover, there are two different status entities (namely, the pull request status and the issue status) that need to be queried to get the succession of events on a pull request.

Commit users: Git allows users to setup custom user names as their commit names. The prevailing convention is that users use their email as their commit names; this is not a strict requirement though. By matching the commit email to the email the user has registered with at Github, it is possible for Github to report the same username across all entities (commits, issues, wiki entries, pull requests, comments etc) affected by a user. GHTorent relies on the git user name resolution to link an entry in the users table to an entry in the commits table. If the commit user has not been resolved, for example because a commit user is not a Github user or the git user's name is misconfigured, GHTorent will create a fake user entry with as much information as available. If in a future update, the resolution does take place, GHTorent will attempt to replace the fake entry with the normal entry. Despite this, there are several thousand fake users in the current dataset.

Pull requests merged outside Github: Although Github automates the generation and submission of patches among repositories through pull requests, those need not be merged through the Github interface. Indeed, several projects choose to track the discussion on pull requests using Github's facilities while doing the actual merge using git. This behaviour can be observed in projects where an usually big number of pull requests are closed without being reported as merged. In such cases, we can deduce that a pull request has been merged by checking whether the commits (identified by their SHA `id`) appear in the main project's repository (through a metadata query). However, this heuristic is not complete, as several projects use commit-squashing or even diff-based patching to transfer commits between branches, thereby loosing authorship information.

Issue tracking is open ended: Repository mining for bug tracking repositories is greatly enhanced, if records are consistent across projects. This is why most studies have been carried on Bugzilla data, which offers a good default set of properties per bug and little opportunities to customize the bug report further. On the other hand, Github's bug tracker only requires a textual description to open a bug. Bug property annotations (e.g. affected versions, severity levels) are delegated to project specific labels. This means that characteristics of bugs cannot be examined uniformly across projects.

Changing data formats: As Github is in active development, the provided data formats and API endpointss are moving targets. During the lifetime of the project, the commit entry schema changed twice, while the watchers entity has been renamed to stargazers. We try to follow the developments that affect the generation of our relational schema only; so far, no modification was necessary.

Some events may be missing: Malfunctions in the mirroring system (software or network) can result in some parts of the data that are missing. In principle, apart from events, all missing data in GHTorent can be restored (by replaying the event log or using the `ght-retrieve-repo` script) provided that the original data have not been deleted from

Github. In the case of missing events, the current Github API does not permit retrieving more than the 300 newest per repository. On busy projects, this is less than a day's worth of event log. Known periods of missing events are several days at the beginning of March 2012, when an error to the event mirroring script went unnoticed, and from mid October 2012 to mid November 2012, when we were trying to adapt GHTorent to the newly imposed requirement for authenticated API requests.

REST queries return modified results: Some REST API calls return slightly modified results if they are queried in different time moments. We have noticed this behaviour in the `created_at` timestamps in several entities. In cases where the field is used as part of a primary key, this might lead to duplicated records. Currently, this behaviour affects the pull_request_history and issue_events.

IV. Research Opportunities

The GHTorent dataset is a rich source for both software engineering and big data research; we outline some research opportunities emerging from this data set below:

1) Unified developer identities: MSR researchers have long faced the problem of disjoint developer identities when attempting to do research across projects and data sources. The GHTorent dataset offers combined source code, source code management, code review, issue and social data using a single developer identity. Researchers can thus track developer actions across projects (e.g. developer migrations) and combine them in novel and interesting ways.

2) Software ecosystems: In Github, project ecosystems are created through forking, sharing of developers and dependency based linking of components. The GHTorent dataset has rich, timestamped information about projects and their forks, which can be easily augmented with library dependency information by automatically browsing related projects.

3) Network analysis: Several networks are being formed on Github, for example project networks through forks, developer networks through participation to common projects, social networks through following other users and watching repositories. Network analysis can either be targeted, for example exploring project community formation dynamics, or abstract by investigating the structure and stability of formed networks to create predictors of future behavior network behavior.

4) Collaboration and promotion: Researchers often ask questions regarding the collaboration tactics of developers and membership promotion strategies in OSS project organizations. The GHTorent dataset, provides timestamped data (albeit since the beginning of the GHTorent project only, see Challenge III above) to investigate how small contributions (known as "drive-by commits") and project forking leads to developer and project collaboration and promotion of an external developer to a team member.

5) Replications of existing studies: A common theme in current software engineering research is the lack of replications or the mediocre replicability of existing works. The GHTorent dataset offers an opportunity to replicate existing work and scale research to many projects, as the dataset is homogenized across several thousand projects, which can be queried for specific characteristics (e.g. programming language, team size, presence of external collaborators etc).

6) An extensible dataset: While GHTorent is covering all public Github entities, it does not include advanced ways of linking them yet. For example, projects can be linked by means of dependencies in their build systems, while commits may be linked with issues by searching for issue numbers in commit messages. The design of the data update process in GHTorent makes such extensions possible: database changes are tracked in a systematic way through migrations, while command line clients that exploit the distribution infrastructure are trivial to develop. Collaborating researchers can thus extend the dataset with custom analyses and data linking facilities.

V. Related Work

Similar to GHTorent is the Github Archive project [3]. Both projects mirror Github's public event timeline. The timeline sources are different; while GHTorent uses the official Github API timeline, Github Archive parses the data used to create the corresponding Github web page. Due to the differences in the two API endpoints, Github Archive's event dataset is richer. However, GHTorent also retrieves the data linked from the event timeline, which allows it to go back in history; in fact, while Github Archive's data starts in February 2012, GHTorent's dataset can and has been extended for certain projects to 2008. In addition, the GHTorent toolset allows for retrieving the full history for a single project and the full list of actions for a single developer, which may make it more appealing to the MSR community.

VI. Conclusions

In this work, we presented the GHTorent dataset and suite of tools, analyzed the mirroring process and outlined the limitations of the current data. The provided dataset has a strong potential for providing interesting insights in areas including but not limited to community dynamics, global software engineering and distributed collaboration. We are actively seeking contributions that will enhance the collected dataset's utility to the research community. More information can be found at http://www.ghtorrent.org.

The source code for the project can be obtained at https://github.com/gousiosg/github-mirror.

Acknowledgements

This work is funded by Marie Curie IEF 298930 — SEFUNC.

References

[1] G. Gousios, M. Pinzger, and A. van Deursen, "An exploration of the pull-based software development model," Mar. 2013. Submitted to the ACM symposium on the Foundations of Software Engineering 2013.

[2] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *MSR '12: Proceedings of the 9th Working Conference on Mining Software Repositories* (M. W. Godfrey and J. Whitehead, eds.), pp. 12–21, IEEE, June 2012.

[3] I. Grigorik, "The Github archive," Mar. 2012. Online, accessed Feb 2013.