Overview of Data Source Used in Software Engineering Studies

Anbarasi Manoharan
North Carolina State University
amanoha2@ncsu.edu

Mathioli Ramalingam
North Carolina State University
mramali2@ncsu.edu

ABSTRACT

This paper aims at discussing how GitHub can be a beneficial source of data for software engineering studies. This report cites multiple research papers to explain the promises and perils of using git as a data source, the workings of decentralized source code management(GitHub in particular), the nature of real data and how imperfect solutions are good enough, how GHTorrent solves the perils of mining GitHub, the more improved GHTorrent ie. the lean GHTorrent. We have also discussed about a few researches which became possible because of availability of this form of data.

Keywords

Decentralized source code management (DSCM), Git, GitHub, GHTorrent, Lean GHTorrent

1. INTRODUCTION

A common requirement of many empirical software engineering studies is the acquisition and curation of data from software repositories. 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 both the commits to the projects' repositories and events generated through user actions on project resources. GHTorrent provides a scalable off line mirror of GitHub's event streams and persistent data, and offer it to the research community as a service.

We are explaining the various factors that influenced the usage of Git/ GitHub as a source of data for software engineering studies, the ecosystem of GitHub, how this large data source can provide imperfect answers which are acceptable, the rationale behind choosing GHTorrent, the more new and modified lean GHTorrent which provides data-on-demand.

2. Data from DSCM

Decentralization has become the major drive in software development. This can be observed by development of a particular project happening at different locations simulataneously. It does not stop there. The concept of not having a centralized source code management has bloomed so much that DSCM has become a part of developers' personal usage as well.

2.1 Mining Git

The number of software projects using DSCMs has increased, and continue to do so. Git is a rich source for software engineering research [4]. The promises and perils of mining git is given as a quick overview below.

2.1.1 Promises

- The personal experiments and false starts are not lost in DSCM(Decentralized Source Code Management). Also, since any developer on a git project can make their own repository publicly accessible, it is possible to recover more history, including work in progress and work that never makes it into the stable codebase.
- Git facilitates recovery of richer project history through commit, branch and merge information from multiple repositories.
- Git has private logs to handle a lot of perils stated below.
- Paper trails can be achieved by "signed-off-by".
- Git explicitly records authorship information for contributors who are not part of the core set of developers.
- In Git, all metadata, notably history, is local.
- Git tracks content, so it can track the history of lines as they are moved or copied.
- Git is faster and often uses less space than centralized repositories.
- Most SCMs such as CVS, SVN, Perforce, and Mercurial (Hg), can be converted to git with the history of branches, merges and tags intact.

2.1.2 Perils

- The nomenclature is conflicting between centralized and DSCM. a) similar actions have different commands; and b) shared terms can have different meanings.
- DSCM has implicit branching, which can be confusing for CSCM developers.
- Git has no mainline, so analysis methods must be suitably modified to take the DAG (Directed Acyclic Graph) into account.
- Git history is revisionist: a repository owner can rewrite it. Rebasing allows modifying the history.

- Determining the branch on which commit was made is not always possible.

- It is not always possible to track the source of a merge or even determine if a merge occurred.

- The accessible data may only contain commits that are success selected.

2.2 GitHub

GitHub has emerged as a popular project hosting, mirroring and collaboration platform. GitHub provides an extensive REST API, which enables researchers to retireve both the commits to the projects' repositories and events generted through user actions on project resources.

GitHub is hosting almost 4 million repositories from 1.5 million users, receiving more than 70,000 commits per day. As a result, the processing, storage and network capacity requirements to process entire stream can be quite steep.

An overview of the GitHub's data schema and API can be found at Figure 1 [11]. A repository is the equivalent of what we call as project. A *user* can sign into GitHub and can create a *repository* under his account. He can manage other people's repositories by participating to an *organization* or *team*. A user can *commit* changes to the repository; these may originate from another author as well.The feature request or problem with the repository can be associated with an *issue*. A collection of issues that must be closed within a target date can be referred as *milestone*. A *pull request* associated with the repository presents some changes that a user can integrate into the repository. Both issues and pull requests can have *comments*. Users can follow each other and see followed user's activity.

GitHub provides a REST API to navigate between the various entities. Figure 1 lists the navigation paths and corresponding URLs as edges between the graph's elements. Other links, such as an issue's assignee, a commit's author, or a milestone's creator, are embedded directly within the returned elements and can be identified in the Figure through their corresponding data types.

In Figure 1, a commit's user, author, and parents appear on a grey background to signify that the philosopher's stone of software repository miners, namely the relationship between commits and other repository data, is not always directly available on a commit's fields.

2.3 Ecosystem of GitHub

Figure 2 shows the full Dependency Network, though for visibility we only display nodes with degree of 3 or greater. The largest connected component (or giant component) is easily identified as the well-connected subgraph appearing in the center of the graph. As visible on the graph, most of the nodes isolated from the giant component are connected to only a small number of nodes. In fact, 75% of nodes not in the giant component are connected to only one other node.

Reference coupling denotes the cross-references to other repositories appearing in GitHub comments indicating the existence of a technical dependency [5].

2.3.1 Research questions

Research Question 1: Does cross-references to other projects in issue, pull request, and commit comments indicate the existence of a technical dependency between the two projects?

Most of the cross reference comments (90%) were classified as technical dependencies. The remaining 10% does not have enough details for proper classification. Two types of dependencies were identified.

Dependency between the two projects. The most common type of dependency found was the direct technical dependency between two projects. For example, the issue created in one project depends on the fix/update in another project.

Both projects depend on third project. There were some cases where the comments describes the dependency on third project that is not cross-refernced.

Research Question 2: What ecosystems exist across GitHub-hosted projects and what is their structure?

High modularity indicated dense connection within communities and sparse connection between communities. High modularity was obtained proving close technical dependency than a random graph.

Research Question 3: Do the project owners' and contributors' social behaviours align with the technical dependencies?

The social behaviour of project contributors is not aligned with project dependencies. Contributor follower network communities do not have one central project and the network is much more densely connected.

2.3.2 Properties of an ecosystem

- Ecosystems revolve around one central project

- Predominant type of ecosystems is software development support

- Ecosystems are interconnected

2.4 Perils of mining GitHub

A more recent research [14, 15] discusses in depth the following perils which might be important factors in mining GitHub.

- A repository is not necessarily a project. A project is typically a part of a network of repositories: at least one of them will be designated as central, where code is expected to flow to and where the latest version of the code is to be found.

- Most projects have low activity in terms of commits.

- Most projects are inactive.

- A large portion of projects are not used for software development activities.

- More than two thirds of projects have only one committer which makes the repositories meant to be for personal use.

- Many active projects do not conduct all their software development activities in GitHub.

- Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.

- Merges only track successful code: If the commits in a pull request are reworked (in response to comments), GitHub records only the commits that are the result of the peer review, not the original commits.

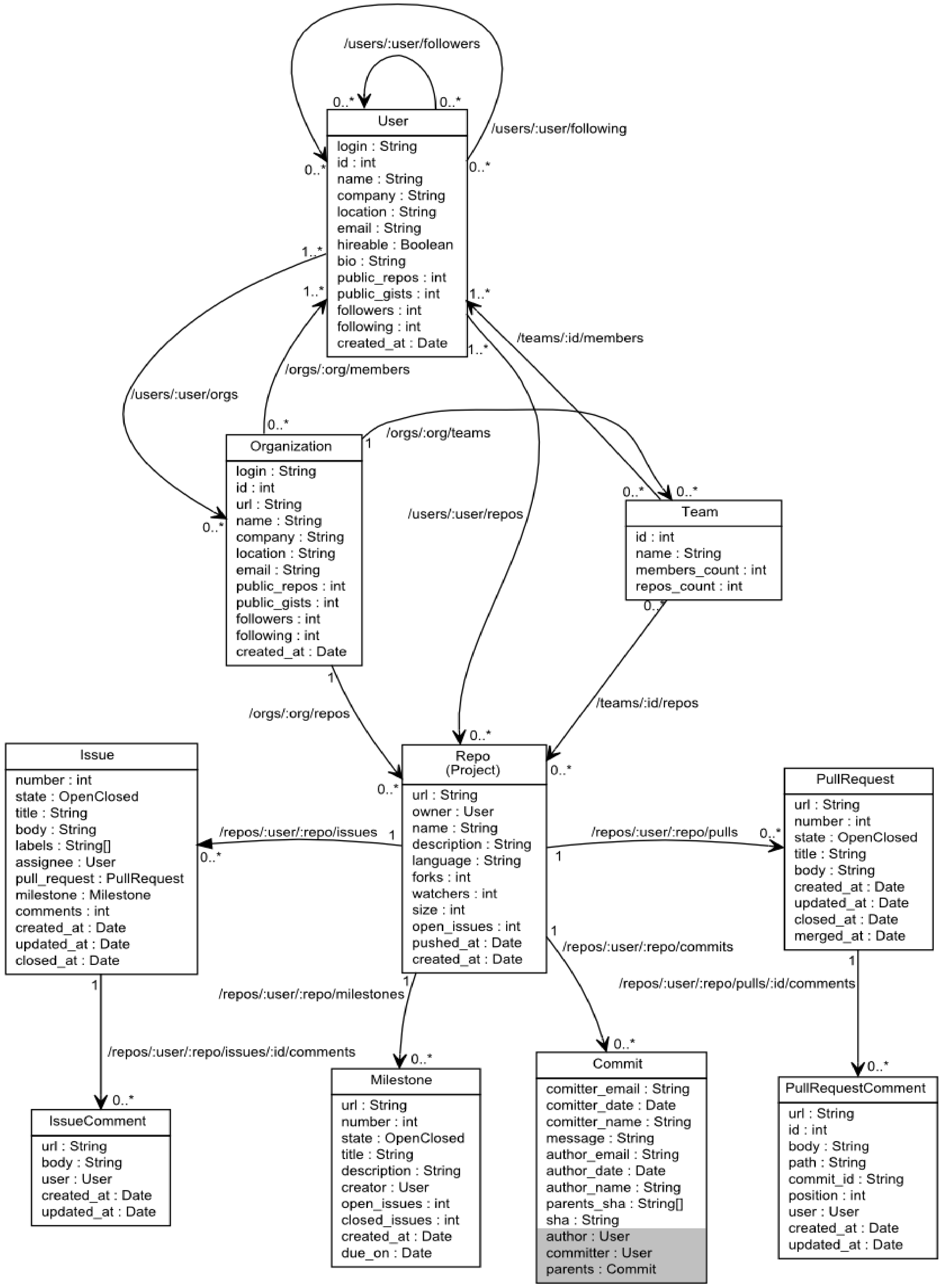


Figure 1: An overview of GitHub's data schema and API

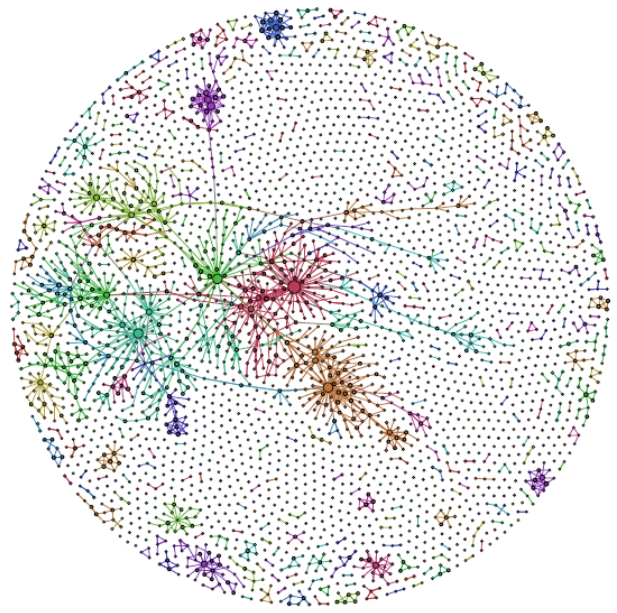


Figure 2: All GitHub projects with cross-references.

- Only pull requests merged via the "Merge" button are marked as merged. But pull requests can also be merged via other methods, such as using git outside GitHub; in those cases, the pull-request will not appear as merged.
- Not all activity is due to registered users: The activity in GitHub repositories is sometimes due to non-users; in some cases, the activity of a user is not properly associated with her account.
- Only the user's public activity is visible: Approximately half of GitHub's registered users do not work in public repositories.
- GitHub's API does not expose all data: The GitHub API exposes either a subset of events or entities, or a subset of the information regarding the event or the entity.
- GitHub is continuously evolving: GitHub continues to evolve and it has changed some features and provided new ones. Similarly, the projects evolve and are capable of changing their own history.

2.5 Too much data?

Pat Helland in his paper [13], explains how the research data should be migrating from locked to unlocked data. This is because real world data is continuously changing and a locked data storage like SQL is no longer sufficient. Humungous data accumulated nowadays are unstructured, the semantics keeps changing if it exists, the answers are not accurate.

Huge scale has implications on the consistency semantics of the data. NoSQL cannot handle transactions across all the data. The data undergoes DDL transformation to be saved in a database, but unlocked data is supposed to be immutable.

When considering the large and scalable applications that are gravitating to the NoSQL model, we must take into account the semantics of the derived data that inevitably accompanies them. The application probably wants a simplified and distilled view of the aggregated data. This view is inevitably a lossy prism that looks at a subset of the knowledge. But only by accepting the inevitability of the loss, we can look for ways to gain business value from the resulting tasty stew.

These lossy answers are not accurate or perfect. But when there is too much data, then a 'good enough' answer is good enough.

Yet another form of knowledge is gathered by patterns and inference. By studying the contents of data, it is sometimes possible to infer that two seemingly disparate identities are really the same. This happens when looking at a catalog of shoes from different merchants, for example. The lack of an industry-wide identifier such as a UPC for shoes means they are particularly subject to erroneously interpreting two shoe descriptions as independent items.

Inference engines look at the relationships between identities (and attributes of the identities) to recognize that two ostensibly different items are the same. Once you realize two identities are the same, new relationships are recognized by merging the two identities into one. Inference engines are constantly appending new information to the data gleaned from the Web and partner companies. The information can be gained from inferences where we can find two disparate things are actually the same.

Through this we can understand the large scale availability of data and how much it can influence in the inference of any experiment.

3. GHTorrent - An Introduction

GHTorrent aims to create a scalable off line mirror of GitHub's event streams and persistent data, and offer it to the research community as a service.

GitHub is hosting almost 4 million repositories from 1.5 million users, receiving more than 70,000 commits per day. As a result, the processing, storage and network capacity requirements to process entire stream can be quite steep. GHTorrent address this issue through the distribution of the data collection and its replication among research teams. They collected commits through GitHub's RSS feed and its commit stream page when the RSS feed became unavailable. Then they changed the collection to be based on events and expanded the collection operation to use event data as the starting point.

Patterns - There can be period of inactivity while processing these huge amount of datas and there should be some kind of monitoring to identify this. Here, they have employed a mechanism to monitor the daily event retrieval activity in order to make sure that such problems will not remain undetected for more than a few hours. When we retrieve huge amount of data, it is best to distribute the data among different system to manage the network and storage capacity and implement a mechanism to consolidate the data.

3.1 Design

GHTorrent is designed to cope with scale through the distribution of the data collection and its replication among research teams. GHTorrent is designed to use multiple hosts for collecting the GitHub static data and dynamic events, thus overcoming the rate limit restrictions. In the future a

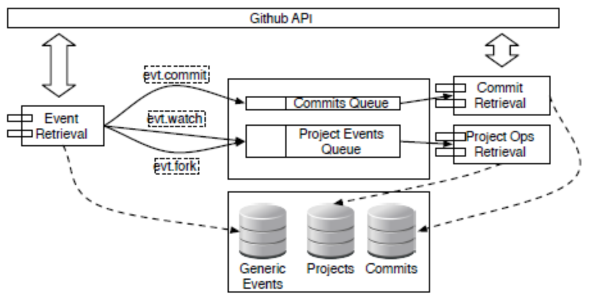


Figure 3: Data retrieval system architecture

network of co-operating research teams, where each of which uses its own infrastructure to mirror diverse parts of the GitHub data, can cooperate to increase the data's coverage. The collected data are then distributed among the research teams in the form of incremental data dumps through a peer-to-peer protocol. The data's replication among those downloading it, ensures the distribution mechanism's scalability [11, 7].

Retrieval of data is performed in stages, which are connected through the pipeline paradigm over a message queue. The retrieved data are stored in a document-oriented NoSQL database, to accommodate future changes to the data returned by GitHub. An overview of the mirroring architecture is presented in Figure 3.

The data retrieval is preformed in two phases: in the event retrieval phase, a crawler queries the GitHub events API, and stores new events in the events data store. It then sends the payload of each new event as a topic exchange message to the system's message queue, along with an appropriate routing key. The topic exchanges enable clients to declare queues selecting the required messages by applying regular expressions on message routing keys. Therefore, during the event processing phase, clients setup queues and select interesting messages for further processing; for example, the commit retrieval client selects all messages routed as commits and then queries GitHub for the commit contents. Similarly, the project details retrieval agent can setup a queue that collects all watch and fork events to query GitHub and update the project's state in the local database mirror.

3.1.1 Limitations of GHTorrent [9]

- **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 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 username 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 endpoints 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 case of missing events, the current GitHub API does not permit retrieving more than the 300 newest events 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. They 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.

3.1.2 *Opportunities*

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

- **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.

- **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.

- **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.

- **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.

- **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).

- **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.

3.2 Lean GHTorrent

To facilitate studies of GitHub, GHTorrent [11], a scalable, queriable, offline mirror of the data offered through the GitHub REST API. GHTorrent data has already been used in emprical studies (e.g., [10, 19, 22]). In [12], they have presented a novel feature designed to offer customisable data dumps on demand. The new GHTorrent data-on-demand service offers users the possibility to request via a web form up-to-date GHTorrent data dumps (in both MySQL and MongoDB formats) for any collection of GitHub repositories.

The existing limit of 5000 requests per hour is less than the actual rate of 8,300 an hour. A single account is not sufficient to get the image. GHTorent overcomes this using memoization [9].

Memoization is done and hence the data is cacheable. The result has dependency resolution that can fail and that can lead to failure in retrieving the result on the whole.

Essentially any study of a restricted collection of GitHub repositories can be carried out using the lean GHTorrent, with advantages such as flexibility in selecting the repositories or reproducibility of the results. We envision, for example, use cases in which researchers interested in mining GitHub data start off by using the in-browser interface to select a number of GitHub repositories matching their research goals. Then, lean GHTorrent can be used to retrieve data for those repositories. In Lean GHTorrent [12], customisable data dumps are being provided to user. Here decentralization is achieved with worker-queue model found at Figure 4.

The architecture of the new GHTorrent data-on-demand service consists of two loosely coupled parts: a web server that handles data requests from users and the GHTorrent server that performs the data extraction. The two servers communicate via messaging queues.

The interaction between the different subcomponents of the web and GHTorrent servers is illustrated in Figure 4. First, users specify thier requests for data by filling in the web form at *http://ghtorrent.org/lean* ①. A request listener validates each request and records metadata about its owner, payload, timestamp and status (completed, in progress) in a relational database. Then, for each GITHUB repository part of the request, the listener posts a message to the queue ② containing the request identifier, tiestamp and repository.

On the GHTorrent server side, a dispatcher listens in on the requests queue ③ and interprets the messages received as follows. First, if the message refers to a request for which no previous messages have been received, a new shared relational database is created to collect metadata for the repositories part of this request ④. This database has the same schema as the original GHTorrent MySQL database [11], reproduced for completeness. Then, for each message (repository) referring to the same job, a retrieval worker is instantiated having as parameters the repository being requested, details for connecting to the job database, and the timestamp of the request ⑤.

Retrieval workers run in parallel and make extensive use of caching. If the main GHTorrent MongoDB databse already contains data for this repository, then the shared job database is populated with metdata for this repository extracted from the main GHTorrent MongoDB database ⑥. Otherwise, both the main GHTorrent database and the job database ⑦ are updated with data freshly extracted from the GITHUB API. This data collection process, again designed as a decentralized process, with decentralization mediated using a similar *worker queue model*, was described previously [9]. Once a retrieval worker finishes, it posts a message to the responses queue ⑧ signalling the completion of its task.

On the web server side, a response listener handles incoming response messages (one for each repository in each request) ⑨ and update the status of the job in the requests database. When "task complete" messages have been received for all the repositories part of request, data dumps are being created from both the job database (MySQL, having the GHTorrent schema) and the main GHTorrent database (MongoDB, only collections - groupings of MongoDB documents - relevant for this request are extracted).

Finally, the request owner is notified via email that her job has completed and the requested data dumps are available for download at a given URL. Here, there is a flexibility in selecting the repositories. It also ensures data dumps are available for reproducing results.

3.2.1 Advantages

- It lowers the "barrier for entry" as the data-on-demand provides easier access to a specific set of GitHub data
- Snapshots of data are available to ensure replicability
- Research show researchers prefer data dumps over online querying

3.2.2 Limitations

Currently, lean GHTorrent has a number of limitations.

- The data dumps offer only first order dependencies e.g., contributors to a repository and their followers, but not followers of these followers.
- The dumps can take days to complete depending on request size and server load.
- No recovery actions in case of errors are currently implemented potentially leading to incomplete dumps, e.g., if GitHub fails to answer an API request. Researchers using lean GHTorrent data are advised to check the integrity of the data dumps themselves and, in case of incomplete data, use the ght-retrieve-* scripts in the main GHTorrent distribution to fill in the data holes, or request data from lean GHTorrent again.
- To limit the load on GHTorrent servers, requests to lean GHTorrent should not exceed 1000 repositories. Researchers interested in mining more than 1000 repositories for a given study can still use the complete GHTorrent dumps.

For large scale, lean GHTorrent is not a good option as there is GHTorrent to satisfy our needs and also this can take forever to complete.

4. SAMPLE USAGE

4.1 Investigating the Geography of Open Source Software through GitHub

A study of the geography of open source software development that looks at GitHub. It was found that developers are highly clustered and concentrated primarily in North America and Western and Northern Europe, though a substantial minority is present in other regions. Code contributions and attention indicate a strong local bias. Users in North America account for a larger share of received contributions than of contributions made. They also receive a disproportionate amount of attention [20, 23].

Mostly the distance determines the collaborators. The contributors are not equally distributed. The main developers are few and are concentrated in a particular area.

Uneven global distributions of open source software developers likely has many reason. The factor can be as simple as availability of internet access and the economic factors. An important contributing factor, however, is likely the locality of the open source practise. The core contributors are homogenous in nature as they share common goals and mental models. The core contributors are fewer in number for a project while there is a wider range for the peripheral community. The data is taken mainly from GitHub which provides a relational data in many cases like following, repository watching, code contributions. Clustering was done to avoid overlapping region names. Since countries/regions were different in size, the division was made that merged countries and also separated at some level to

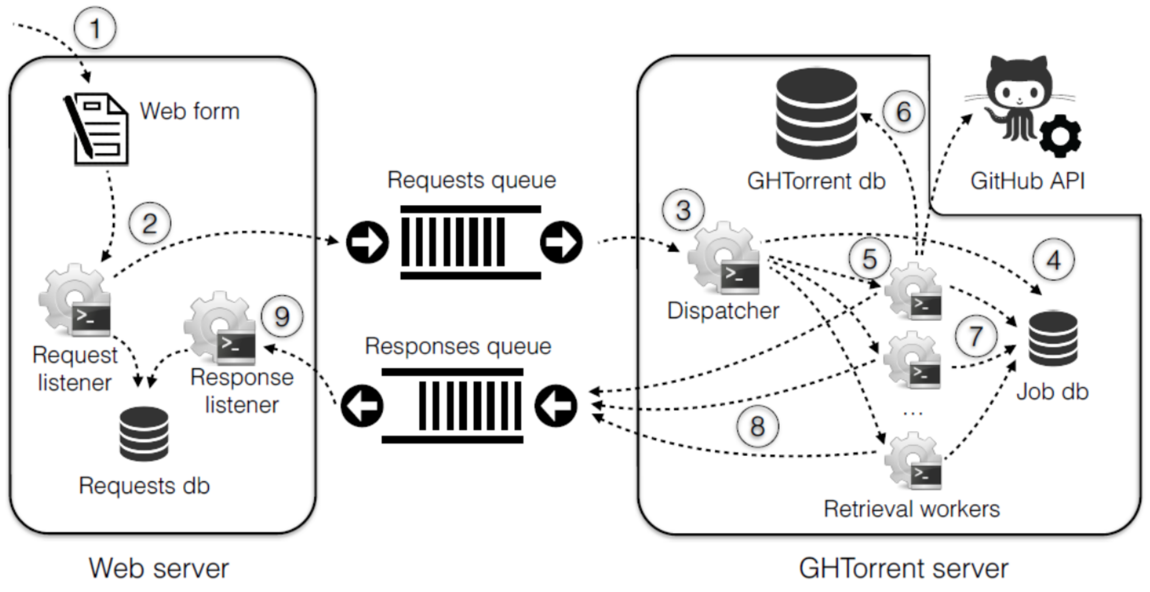


Figure 4: Architecture of GHTorrent data-on-demand service

make more sense. Contributors were mostly belonging to the same clusters.

As shown in Figure 5, North America and W & N Europe account respectively for 43% and 26% of the users. The remaining regions account for 11% or less each. The North America's remains about the same if we only count users for whom we registered at least one code contribution, but rises to 48% when we count at the total number of contributions associated with accounts in North America. This gain comes at the expense of each of the other regions except for Australia and New Zealand. The loss is most substantial for Latin America, which accounts for 6.4% of registered user accounts but only for 3.6% of the contributions. A large part of this loss comes from Brazil, which accounts for 4.2% of users and 2.2% of contributions.

Despite the dominance of North America and W & N Europe, it is important to note that other regions do jointly account for 31% users and 30% of the contributors.

4.2 Understanding the popular users: Following, affiliation influence and leadership on GitHub

Understanding how influence is exerted on social computing platforms is critical for participants and leaders because it impacts their patterns of work, interactions, and knowledge management in collaborative environments [6].

It was observed that the influence depended more on the popularity(people with 500 or more followers) of the user than the contribution by the user. When a rockstar contributes to the project he owns, he invites more contribution. When he contributes in a new project, it guides the followers to the new project too. Earlier there was centralization and hierarchy amongst contributors in GitHub, but no leaders at the top.That's not the case now. Observers and participants are merely followers and not active participants/contributors of any sort. Participants comment on issues, perform pull request etc. and are a part of the project in some way.

It was noticed that the rate of popular users' contribution did not affect the influence they had on the followers.

Research Question 1: Why do GitHub users follow others and who are the most followed users?

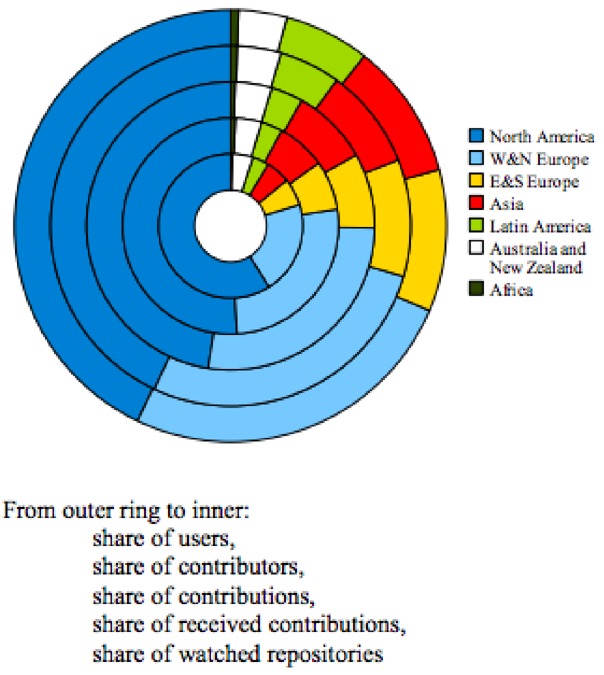
Reasons behind why people follow:

- Getting updates on the project. Following the core contributors enables this.
- Discovering new projects and trends. Staying up-to-date can be achieved this way.
- Learning.
- No benefits.
- Socializing. To follow friends/ co-workers.
- Collaboration. To find opportunities to share code.
- General interests.
- Easy access to people - like a bookmark.

These facts show that influencers can impact easily in the working pattern in GitHub. Followers also believed that the popular people on GitHub are experts. Collaboration is more a perceived benefit than an actual benefit. Finding new projects/ trends likely was not a well-known benefit.

Reason people do not follow:

- No benefit is seen.
- Following project is more useful.
- Busy.
- Using GitHub for personal use.
- Unaware of the feature.



From outer ring to inner:
share of users,
share of contributors,
share of contributions,
share of received contributions,
share of watched repositories

Figure 5: Five participation metrics as a diagram

- Information overload happens.

Research Question 2: Are GitHub users influenced by the users they follow? Popular user actions attract their followers to new projects. Followers are likely to star new projects after a popular user whom they are following does any activity on that project.

Starring a project is powerful - The contribution showed no statistical significance. The more followers a popular user has, the more their followers are influenced by their actions. The activeness a popular user is does not impact their influence.

4.3 Multitasking across GitHub projects

Multitasking has become increasingly common among knowledge workers [8, 17, 18]. While there is some disagreement between studies, it is generally believed that multitasking has non-monotonic effects on individual productivity [1, 3], with plenty of theoretical evidence from Psycology, Management, and Organizational Behavior (e.g., [1, 18]).

Software development has always inherently required multitasking: developers switch between coding, reviewing, testing, designing, and meeting with colleagues. The advent of software ecosystems like GitHub has enabled something new: the ability to easily switch between projects [21].

Developers also have social incentives to contribute to many projects; prolific contributors gain social recognition and (eventually) economic rewards.

Positive effects are attributed to several factors. First, multitasking may increase productivity since inevitable lulls in one project can be filled with tasks on other project. Second, it increases productivity through cross-fertilisation and learning. If knowledge and skills required to work on project are transferable to other projects as well, one may experience decreasing costs of contributing to multiple projects simultaneously, as they are able to realize such knowledge transfers [16].

On the other hand, several theories explain the cognitive processes that account for decreased performance in multitasking conditions, e.g., memory-for-goals [2]. Frequent context-switches can lead to distraction, sub-standard work, and even greater stress. An ecosystem-level data on a group of programmers working on a large collection of projects was collected and models and methods for measuring the rate and breadth of a developers' context-switching behavior. The study gives the following results:

It was found that the most common reason for multitasking is inter-relationships and dependencies between projects. Notably, the rate of switching and breadth (number of projects) of a developer's work matter. Developers who work on many projects have higher productivity if they focus on few projects per day. Developers that switch projects too much during the course of a day have lower productivity as they work on more projects overall. Despite these findings, developers perceptions of the benefits of multitasking are varied.

5. CONCLUSIONS

Multiple sources mention the shift towards DSCM and how researches on software engineering is performed more using GitHub by the day. We have not only observed Git being a good transition for developers, but also as a great source of data that can be explored to drive software engineering to the next stages. The samples described are just a few success stories. However, in every research there was a necessity to validate the results from these data sources, which was not always simple. We find a strong need for simple validation methodologies while using GitHub, that can be applied across most of the software engineering studies.

6. REFERENCES

[1] R. F. Adler and R. Benbunan-Fich. Juggling on a high wire: Multitasking effects on performance. *International Journal of Human-Computer Studies*, 70(2):156–168, 2012.

[2] E. M. Altmann and J. G. Trafton. Memory for goals: An activation-based model. *Cognitive science*, 26(1):39–83, 2002.

[3] S. Aral, E. Brynjolfsson, and M. Van Alstyne. Information, technology, and information worker productivity. *Information Systems Research*, 23(3-part-2):849–867, 2012.

[4] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10. IEEE, 2009.

[5] K. Blincoe, F. Harrison, and D. Damian. Ecosystems in github and a method for ecosystem identification using reference coupling. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 202–207. IEEE Press, 2015.

[6] K. Blincoe, J. Sheoran, S. Goggins, E. Petakovic, and D. Damian. Understanding the popular users: Following, affiliation influence and leadership on github. *Information and Software Technology*, 70:30–39, 2016.

[7] K. K. Chaturvedi, V. Sing, and P. Singh. Tools in mining software repositories. In *Computational*

Science and Its Applications (ICCSA), 2013 13th International Conference on, pages 89–98. IEEE, 2013.

[8] V. M. González and G. Mark. Managing currents of work: Multi-tasking among multiple collaborations. In *ECSCW 2005*, pages 143–162. Springer, 2005.

[9] G. Gousios. The ghtorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236. IEEE Press, 2013.

[10] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[11] G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*, pages 12–21. IEEE, 2012.

[12] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th working conference on mining software repositories*, pages 384–387. ACM, 2014.

[13] P. Helland. If you have too much data, then'good enough'is good enough. *Communications of the ACM*, 54(6):40–47, 2011.

[14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[15] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.

[16] A. Lindbeck and D. J. Snower. Multitask learning and the reorganization of work: from tayloristic to holistic organization. *Journal of labor economics*, 18(3):353–376, 2000.

[17] G. Mark, V. M. Gonzalez, and J. Harris. No task left behind?: examining the nature of fragmented work. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–330. ACM, 2005.

[18] M. B. O'leary, M. Mortensen, and A. W. Woolley. Multiple team membership: A theoretical model of its effects on productivity and learning for individuals and teams. *Academy of Management Review*, 36(3):461–478, 2011.

[19] M. Squire. Forge++: The changing landscape of floss development. In *2014 47th Hawaii International Conference on System Sciences*, pages 3266–3275. IEEE, 2014.

[20] Y. Takhteyev and A. Hilts. Investigating the geography of open source software through github, 2010.

[21] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov. The sky is not the limit: multitasking across github projects. In *Proceedings of the 38th International Conference on Software Engineering*, pages 994–1005. ACM, 2016.

[22] B. Vasilescu, V. Filkov, and A. Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Social Computing (SocialCom), 2013 International Conference on*, pages 188–195. IEEE, 2013.

[23] S. von Engelhardt, A. Freytag, and C. Schulz. On the geographic allocation of open source software activities. *International Journal of Innovation in the Digital Economy (IJIDE)*, 4(2):25–39, 2013.