A Network of Rails

A Graph Dataset of Ruby on Rails and Associated Projects

Patrick Wagstrom
IBM TJ Watson Research Center
Yorktown Heights, NY, USA
pwagstro@us.ibm.com

Corey Jergensen and Anita Sarma
Computer Science and Engineering
University of Nebraska – Lincoln, Lincoln, NE, USA
{cjergens, asarma}@cse.unl.edu

***Abstract*—Software projects, whether open source, proprietary, or a combination thereof, rarely exist in isolation. Rather, most projects build on a network of people and ideas from dozens, hundreds, or even thousands of other projects. Using the GitHub APIs it is possible to extract these relationships for millions of users and projects. In this paper we present a dataset of a large network of open source projects centered around Ruby on Rails. This dataset provides insight into the relationships between Ruby on Rails and an ecosystem involving 1116 projects. To facilitate understanding of this data in the context of relationships between projects, users, and their activities, it is provided as a graph database suitable for assessing network properties of the community and individuals within those communities and can be found at https://github.com/pridkett/gitminer-data-rails.**

***Index Terms*—Data, GitHub, Ruby on Rails, Graph Databases**

I. Introduction

While it is possible to gain significant insights into the mechanisms of software engineering and open source software development by studying a single monolithic project, almost all software projects include portions or the entirety of other software projects. Sometimes these secondary projects implement critical aspects of a project, for example the use of WebKit (an HTML rendering engine) in the Chromium and Safari web browsers. Other times they provide infrastructure for the building and management of the project, such as Maven and Jenkins. Even if a project uses only self-contained code and has no external dependencies it still is influenced by other software projects that a developer has experienced – whether through coding standards or through social norms around project development. No project exists as an island and to succeed, the ecosystem in which the project exists is equally important as the project itself.

During its genesis the open source community quickly realized this fact and projects began to coalesce into communities around open source foundations such as Apache and GNOME. These communities provided the mailing lists, bug tracking, source code management, and web page infrastructures that projects needed to survive [1]. Projects that weren't members of a foundation because of a lack of alignment, personal choice, maturity, or a variety of other reasons, were often hosted on large source code hosting sites, the most dominant of which was SourceForge.

SourceForge was novel in that it allowed a user with a single account to develop on many seemingly unrelated projects. It was possible for the first time to determine the breadth of contribution that a person made across a wide variety of projects that were hosted on SourceForge. However, other critical elements of the open source ecosystem, particularly bug trackers, were not traditionally as cleanly integrated. Therefore, while it was possible to build a network of co-developers between projects, it took significant effort to build anything more robust.

The launch of GitHub dramatically changed the landscape of these hosting sites in a number of ways. The first is providing a unified environment for source code and bug tracking together in the same platform. Second, through the use of forks (creating a copy of the repository for personal development), GitHub makes visible the development that occurs outside of the main development branch, even if a user has never pushed their changes back to the main repository but instead works on the code independently.

GitHub also introduced a novel mechanism for managing code contributions from external users. Previously, projects often relied on ad-hoc patch management through mailing lists, which could result in patches being ignored or misplaced in the triaging process. The Pull Request mechanism within GitHub allows a developer to formally request that their changes be integrated into the main project. This mechanism is much more robust than sending a patch through the mailing list as it treats the Pull Request as a first order entity and allows individuals to comment on them or individual lines of code thus preserving the discussion. This provides a provenance for the code that is integrated into a project and also makes the authors of the code apparent.

GitHub also added two major new "social" features: the ability to star (previously called "watch") a project and the ability to follow users. Starring a project means that when a users visits their dashboard they'll see updates about the projects in their feed. Following a user aggregates that user's activities in their dashboard. Together these allow users to signal their interest in a project (or a developer) without being required to formally contribute to a project.

Finally, GitHub provides an API that allows a relatively easy mechanism for accessing the above information. These APIs allow users to download accurate information while being respectful of GitHub's constraints (e.g. maximum number of requests per hour).

We developed a tool called GitMiner [2] that by using the GitHub APIs downloads information (to the extent allowed)

978-1-4673-2936-1/13/$31.00 © 2013 IEEE

MSR 2013, San Francisco, CA, USA

about a set of projects and users from GitHub and then stores the data in a graph database. GitMiner has a small but growing community of researchers across the world that use and contribute to the tool (currently six institutions). However, even with the availability of GitMiner, downloading the data takes significant time, especially because of the API limits regarding the number of requests allowed per hour. We, therefore, have made available a dataset representing a software ecosystem around the Ruby on Rails project, which includes a sample of more than 1116 associated projects [3].

II. GitMiner

We downloaded the data from GitHub by using GitMiner, which was jointly developed by the University of Nebraska-Lincoln and IBM Research. When given a particular project or a user GitMiner connects to GitHub using the GitHub API and downloads all available information about the project or the user. It then downloads copies of the source code repositories and analyzes all commits made by the users associated with the project. This data is then stored in an underlying graph database, as opposed to a more traditional relational database. This architecture is well suited for understanding networks of dependencies between projects and localized searches of individual user activity.

GitMiner can be thought of as a compliment to the work of Ilya Grigorik on the GitHub Archive [4]. While GitMiner focuses on a single project, user, or sets thereof, and dives as deep as possible, GitHub Archive attempts to poll the GitHub main feed as often as possible for information about recent events. GitHub Archive captures only information about activities in the past few moments, and thus needs to constantly poll GitHub or risk losing data (and even still may lose data when there are unexpected spikes in activity on GitHub). This strategy is useful if one would like to examine the current activity across all of GitHub, in which cases gaps in the data may be acceptable for some periods. However, as a result of this strategy, the only data available through the GitHub Archive is from mid February 2011, the start of the project.

GitMiner, in contrast focuses only on a single project or set of projects and downloads their entire history, including some elements not exposed in the GitHub events stream. However, as GitMiner uses the GitHub APIs it has some constraints, notably among them is the maximum number of API requests per hour, which is currently fixed at 5000 for most purposes. A large dataset can then take slightly over a month in order to download.

III. Data Description and Collection

The datasets we present was collected at the beginning of May 2012 and has since been used for research on a variety of different topics. Although the data presented is approximately a year old, GitMiner has the functionality to update previous data sets with newer data.

The GitHub API is able to return the following elements (see Table I for a summary of the data points) that are captured by GitMiner and presented in this dataset.

- `Repository`: One of the central elements in the data. A repository (e.g. rails/rails) represents a top level project on GitHub and also corresponds to the underlying version control system archive for the project.
- `File`: A single file that resides in a git repository. Usually connected to a `Commit`, but sometimes referenced from an `Issue` or `Pull Request`.
- `Commit`: A single commit in the git version control system. It is used to link an author, set of files, and repository together.
- `Gist`: A small piece of code that a user has posted to GitHub. It is possible to fork and modify a gist, but they lack the rest of the infrastructure provided to projects.
- `Pull Request`: A request by one user that a project "pull" some of their code into the main branch.
- `Pull Request Marker`: A reference to a portion of a `Pull Request`. Usually used for making comments about a `Pull Request`.
- `Issue`: An item in GitHub's issue tracker.
- `Issue Event`: An event applied to an `Issue` in GitHub's issue tracker (e.g. open, close, etc.).
- `Label`: A tag that can be applied to an issue for the purpose of managing categories of issues.
- `Milestone`: A tag that can be applied to an issue for the purpose of managing the software development process (e.g. scheduling a issue to be addressed during a sprint).
- `Comment`: A message written by a user in reference to an `Issue`, `Pull Request`, or line of code.
- `User`: Another central element in the data. A `User` represents an account within GitHub.
- `Git User`: An individual who is affiliated with commits in a git repository. Because git provides a complete provenance chain a single GitHub user will often show up as multiple different `Git Users`, particularly if they have changed email addresses.
- `Gravatar`: A processed hash of a users email address that can be used to retrieve an image of a user. This is also used to map some `Git Users` to `User` accounts.
- `Name`: The actual name of a user. Often times users will have slightly different names on multiple git commits while retaining the same email address. This mechanism allows for creating consistency across commits.
- `Email`: A single record of an email address. Connects to a `Gravatar` and sometimes a `Git User` or `User` node.
- `Event`: The GitHub API provides the last 200 actions a `User` has performed on the system. For select users (core contributors in Ruby on Rails, see below), this information is stored.

To collect the dataset, we started with the Ruby on Rails project, which is one of the largest and most successful projects on GitHub. We then spidered out to projects in which contributors from Ruby on Rails were also participating. To keep the dataset manageable, when spidering out we only consider participation by "core contributors" from Ruby on Rails; where core contributors are defined as users who are in the top 20% of code commits or issue management. We then identified (other) projects that these users were contributing to (had commit access), projects that they starred ("watched the project"), or oth-

other activities and collected the data on these projects. This method led to a dataset of 1116 repositories [3].

Although the GitHub API provides robust mechanism to retrieve most data, there were some locations where identification was difficult. The matching of commits to the GitHub accounts of their authors was one such challenge. Many users chose not to make their email addresses public, in which case GitHub returns only the md5 hash of their address that can then be used to obtain their Gravatar. Calculating the md5 hash of the email address for every commit allowed us to link up many of the commits with the corresponding GitHub user, but not all of them. In particular, many repositories that started as local repositories without a properly configured git client have inconsistent user email addresses. In these cases, a perfect match with another name that had committed to the same repository was used to link these commits together. However, sometimes even this was not enough. We finally discovered that in some cases it is possible to obtain additional information about alternate names of users and email addresses by examining the event history of users, their pull requests, and comments, all of which can be linked to commits in a git archive. Using these mechanisms we are reasonably certain that users are properly associated with their contributions.

TABLE I. OVERVIEW OF DATA

Dates Collected		May 5 – May 30, 2012	
Dataset Size		**8.4GB**	
Total Nodes		**4800596**	
Total Edges		**18455346**	
Seed Projects		**1116**	
COMMENT	194714	ISSUE	86995
COMMIT	862349	ISSUE_EVENT	154977
EMAIL	66003	LABEL	1714
EVENT	1301734	MILESTONE	145
FOLLOWER	739848	NAME	27655
FOLLOWING	787545	PULL_REQUEST	41793
FILE	420579	PULL_REQUEST _MARKER	67764
GIST	24464	REPOSITORY	1016672
GIT_USER	33300	USER	286000

IV. ACCESSING AND MANIPULATING THE DATA

In contrast to relational databases that utilize SQL, the data in this dataset and all datasets produced by GitMiner are stored in a graph database. More specifically, this dataset is provided as a Neo4j database that can be copied between machines and loaded into an embedded Neo4j instance, Neo4j server, or a variety of other tools that can expose graph databases, such as Rexster, which provides REST interfaces to graph databases.

Similar to various document databases such as CouchDB and MongoDB, a graph database is schemaless. It consists of a set of elements, each of which may either be a node or an edge. An edge connects two nodes and is always directed and has a (required) label to denote the type of relationship the edge expresses. If a pair of nodes has multiple different types of relationships then multiple edges with different labels may be used between the nodes. The full overview and distribution of edges and nodes is provided with the dataset.

In most cases graphs are accessed using query languages that execute a traversal. Instructions are provided that set the node or collection of nodes to begin at, specify a series of edges to traverse, and then return a collection of values back to the user. For example, the Neo4j database has a built in traversal language called Cypher and Tinkerpop provides a graph database agnostic language called Gremlin. While both are sufficient for traversing the data, for the purposes of describing this dataset we provide instructions using Gremlin [5].

This first snippet of code opens up a connection to the embedded Neo4j database and then retrieves the vertices that represent Ruby on Rails and Jose Valim, a core contributor to Ruby on Rails. For the purpose of this example we're not providing any special tuning to Neo4j, so initial traversals are much slower until the database cache has been primed.

```
gremlin> g = new Neo4jGraph("rails.db.20120505")
==>neo4jgraph[EmbeddedGraphDatabase
[/Users/msr/rails.db.20120505]]
gremlin> rails = g.idx("repo-idx"). \
gremlin> get("reponame","rails/rails").first()
==>v[1]
gremlin> jv = g.idx("user-idx"). \
gremlin> get("login","josevalim").first()
==>v[30]
```

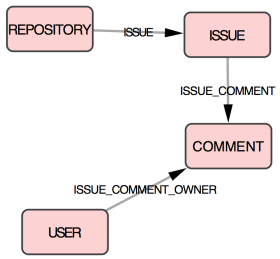

Fig. 1. Traversal to Identify Comment Owners from a Repository

Next we'd like to actually look at some of the relationships, for example, the number of comments on an issue authored by Jose Valim. In the database a REPOSITORY is connected to an ISSUE with an outgoing edge labeled ISSUE. ISSUES are connected to ISSUE_COMMENTS with an outgoing edge labeled ISSUE_COMMENT. Here we first count the number of issues filed against rails in the dataset and then count the number of comments on those issues, and finally we count only the comments on issues that were authored by Jose Valim. The general pattern of the traversal and the direction of the edges can be seen above in Fig. 1. and in the code snippet follows.

```
gremlin> rails.out("ISSUE").count()
==>6324
gremlin>
rails.out("ISSUE").out("ISSUE_COMMENT").count()
==>24310
gremlin> rails.out("ISSUE").out("ISSUE_COMMENT"). \
gremlin> in("ISSUE_COMMENT_OWNER").filter{it==jv}. \
gremlin> count()
==>1560
```

Next, we find out the number of commits by Jose Valim. As explained earlier we use a combination of email address and Gravatar to determine this information. This traversal can run in both directions, as shown below, simply by changing the starting vertex and the direction of the edges. A visual representation of the traversal is shown in Fig. 2.

```
gremlin> rails.in("REPOSITORY").out("AUTHOR"). \
gremlin> out("EMAIL").out("GRAVATAR_HASH"). \
gremlin> in("GRAVATAR").filter{it==jv}.count()
==>2316
gremlin> jv.out("GRAVATAR").in("GRAVATAR_HASH"). \
gremlin> in("EMAIL").in("AUTHOR"). \
gremlin> out("REPOSITORY").filter{it==rails}.count()
==>2316
```

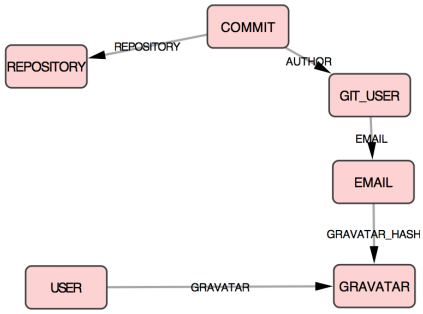

Fig. 2. Traversal to Identify Commits

As a final example, we obtain a list of all the repositories that Jose Valim has committed code to. One of the tricky things about this query is that many repositories are forks of the main rails repository, so this will result in numerous extra repositories for which there may have been no actual action (although, there are serious caveats about this as described by Bird et al. [6]). To accomplish this we look only for repositories that have the property `isFork` set to `false`, remove duplicates with the dedup option, and then just return their names.

```
gremlin> jv.out("GRAVATAR").in("GRAVATAR_HASH"). \
gremlin> in("EMAIL").in("AUTHOR"). \
gremlin> out("REPOSITORY").has("isFork",false). \
gremlin> dedup().reponame
==>rails/rails
==>erlang/otp
==>brynary/webrat
==>jm/rails_upgrade
==>cassiomarques/booleanize
==>elixir-lang/elixir-lang.github.com
==>elixir-lang/elixir
==>ianwhite/orm_adapter
==>plataformatec/devise
==>plataformatec/has_scope
==>lifo/docrails
==>dolzenko/windows_protocol_handlers
```

V. Advantages and Disadvantages of Approach

As Grigorik's work with GitHub Archive has shown, managing extremely large datasets is problematic. Typical relational database systems tend to break down because of causes such as dynamic schema changes, large volumes of data, and increasing difficulty in managing the various systems required to store and query large datasets. We realized in the construction of this dataset and the tooling for GitMiner that most of our queries were localized in small areas around a project or a set of users and that these queries were often replicated across a large number of projects or users.

GitMiner therefore allows the use of distributed graph databases, such as Titan, to take advantage of this. Different nodes can be automatically placed on servers where we can distribute multiple parallel queries across a cluster of machines. By taking advantage of data locality we can easily run analysis of many projects at the same time with minimal additional effort.

However, the use of graph databases presents unique challenges. The first challenge is that graph databases require a paradigm shift in thinking about relations and queries. The lack of a fixed schema and the networked nature means that developers must first learn to map queries to traversals, which can be difficult.

Second, graph databases require more care to properly scale. Although it is possible to take advantage of data locality and partition a graph across multiple nodes in a Hadoop cluster for parallel data analysis, this can be challenging especially for densely connected network of projects. In these cases a poorly laid out network topology will result in poor performance.

Finally, although not exclusively a problem with Graph databases, for some elements GitMiner only provides a snapshot of the data. This is particularly evident when examining users who have watched/starred a project. The API in this case does not provide the date when a user either stars or un-stars a repository. This makes it difficult to understand the evolution of the social action of starring a repository without taking frequent snapshots of the repository and comparing those snapshots.

VI. Future Extensions to Dataset

GitMiner is designed to seamlessly update datasets when new information becomes available. Using the same configuration file a researcher can download newer data about projects and the new database will automatically reflect the state of the project. Each of the nodes and edges in the database is tagged with a property, `sys_last_updated`, that tells when the dataset was last updated.

Acknowledgment

This work was funded by NSF grants CCF 1016134 and IIS 1110916.

References

[1] German, "Software Engineering Practices in the GNOME Project," in Perspectives on Free and Open Source Software, J. Feller, et al., eds. MIT Press, 2005, pp. 211–226.

[2] P. Wagstrom, C. Jergensen, and A. Sarma. "GitMiner". Available from: https://github.com/pridkett/gitminer. 2013. Last Visited: Feb. 8, 2013.

[3] P. Wagstrom, C. Jergesensen, and A. Sarma. "Rails GitMiner Dataset". Available from: https://github.com/pridkett/gitminer-data-rails. 2012. Last Visited: Feb. 8, 2013.

[4] I. Grigorik, "GitHub Archive". Available from: http://www.githubarchive.org/. 2012. Last Visited: Feb. 8, 2013.

[5] M. Rodriguez et al. "Gremlin" Available from: https://github.com/tinkerpop/gremlin. Last Visited: February 8, 2013.

[6] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The promises and perils of mining git. *6th IEEE International Working Conference on Mining Software Repositories* (MSR '09). 1-10.