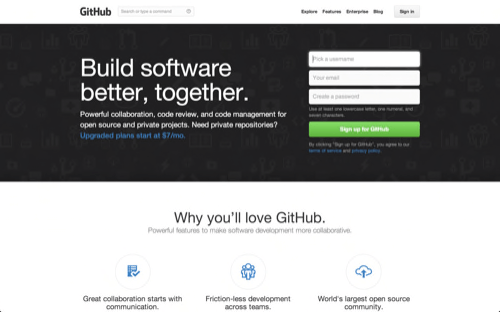
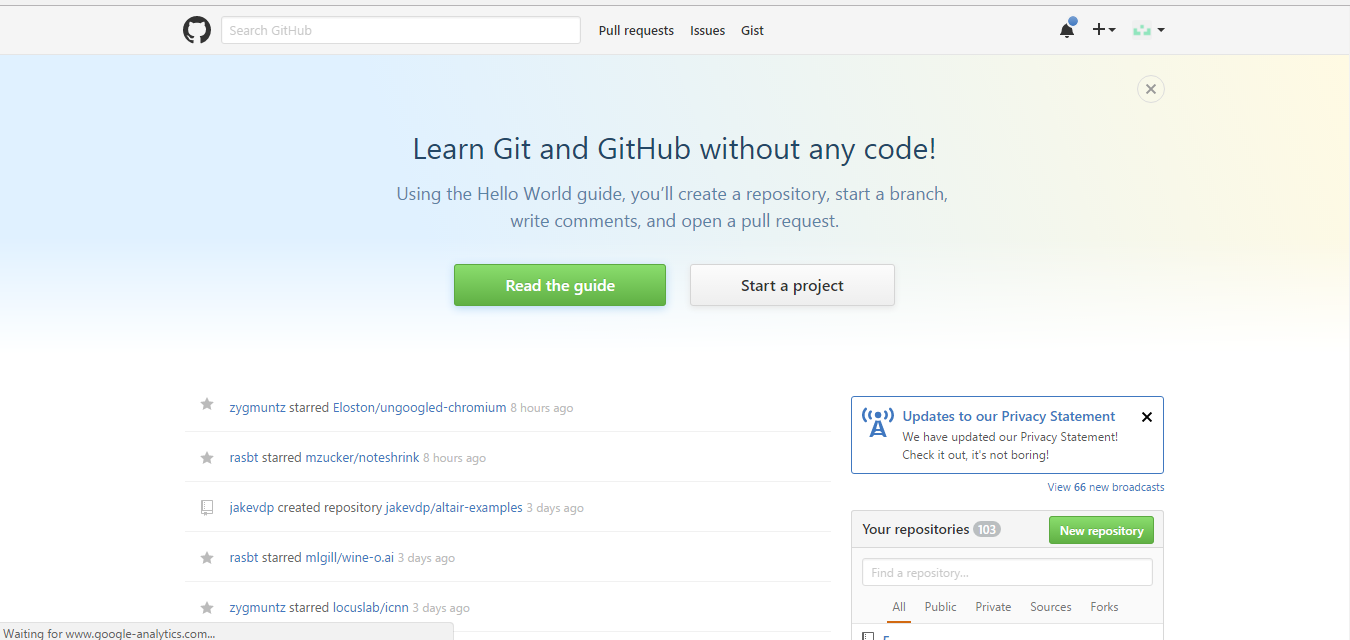
**GITHUB – TOWARDS SOCIAL CODING**



Homepage of GitHub

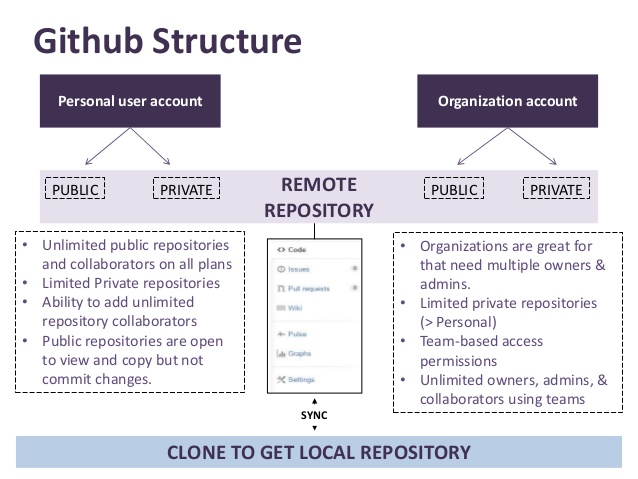


Home of an user

**What is GitHub?**

GitHub is a publishing tool, version control system, and a collaboration platform. GitHub allows you to host remote Git repositories and has a wealth of community-based services.

**Structure of GitHub**



User Accounts on GitHub:

* Personal User Account: Any individual with an email id can open an account with unlimited public repositories. For private repositories, the user needs to buy plans.
* Organization Account: Organizations such as Facebook, Microsoft, Google and many more have well maintained repositories on GitHub. These are public repositories. Organizations can have owners, admins and collaborators based on teams.

**Collaborator**: A collaborator in GitHub is the same as a user with the added benefit of having his own public account. In other words a Collaborator has the ability to set up his own Repositories with their own Wiki sites per Repository. This is quite a nice feature seeing that I can be linked to my Organization for those things that I do on their stuff, but I can also have my own source for articles I write or open source projects that I own. The other really big thing that can be taken into consideration is that as a Collaborator you can join any current open source project and do some development for them. However getting added as a collaborator is not an easy thing.

**Contributor:** A contributor is someone who has contributed to a project by having a pull request merged but does not have collaborator access.

**Repository:** A directory or storage space where your projects can live. Sometimes GitHub users shorten this to “repo.” It can be local to a folder on your computer, or it can be a storage space on GitHub or another online host. You can keep code files, text files, image files, you name it, inside a repository.

**Version Control:** Basically, the purpose Git was designed to serve. When you have a Microsoft Word file, you either overwrite every saved file with a new save, or you save multiple versions. With Git, you don’t have to. It keeps “snapshots” of every point in time in the project’s history, so you can never lose or overwrite it.

**Commit**: This is the command that gives Git its power. When you commit, you are taking a “snapshot” of your repository at that point in time, giving you a checkpoint to which you can reevaluate or restore your project to any previous state.

**Branch:** How do multiple people work on a project at the same time without Git getting them confused? Usually, they “branch off” of the main project with their own versions full of changes they themselves have made. After they’re done, it’s time to “merge” that branch back with the “master,” the main directory of the project.

**Clone**: A clone is a copy of a repository that lives on your computer instead of on a website's server somewhere, or the act of making that copy. With your clone you can edit the files in your preferred editor and use Git to keep track of your changes without having to be online. It is, however, connected to the remote version so that changes can be synced between the two. You can push your local changes to the remote to keep them synced when you're online.

**Fork:** A fork is a personal copy of another user's repository that lives on your account. Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original's author to update with your changes. You can also keep your fork up to date by pulling in updates from the original.

**Issue:** Issues are suggested improvements, tasks or questions related to the repository. Issues can be created by anyone (for public repositories), and are moderated by repository collaborators. Each issue contains its own discussion forum, can be labeled and assigned to a user.

#### Private Repository: Private repositories are repositories that can only be viewed or contributed to by their creator and collaborators the creator specified.

#### User: Users are personal GitHub accounts. Each user has a personal profile, and can own multiple repositories, public or private. They can create or be invited to join organizations or collaborate on another user's repository.

#### Pull Request: Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators. Like issues, pull requests each have their own discussion forum.

#### Merge: Merging takes the changes from one branch (in the same repository or from a fork), and applies them into another. This often happens as a Pull Request (which can be thought of as a request to merge), or via the command line. A merge can be done automatically via a Pull Request via the GitHub.com web interface if there are no conflicting changes, or can always be done via the command line.

## **Merging a pull request**

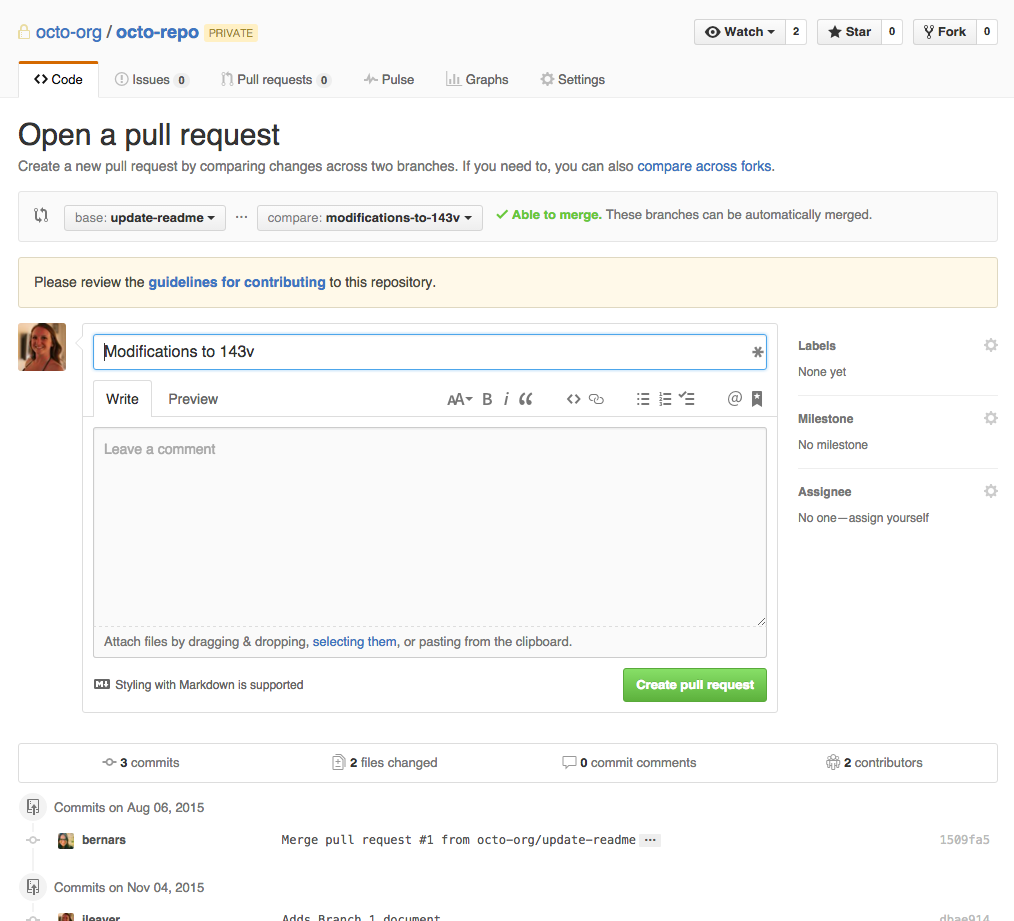
Merge a pull request into the upstream branch when work is completed. Anyone with push access to the repository can complete the merge.

If you decide you don't want the changes in your branch to be merged to the upstream branch, you can also close the pull request without merging. Alternatively, you can ask the pull request creator to allow anyone with push access to the upstream repository to make changes to their pull request. Pull request creators can enable this option for existing and new pull requests. If you'd like to merge changes into a protected branch that has required pull request reviews for pull requests enabled, you'll need to make sure your pull request has met the pull request review policy. In this policy, pull requests must have one approved review and cannot have any reviews that request changes before they can be merged.

Pull requests let you tell others about changes you've pushed to a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before the changes are merged into the repository.

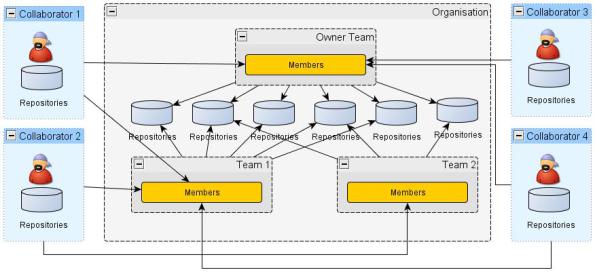
Once you've created a pull request, you can push commits from your topic branch to add them to your existing pull request. These commits will appear in chronological order within your pull request and the changes will be visible in the "Files changed" tab.

Other contributors can review your proposed changes, add review comments, contribute to the pull request discussion, and even add commits to the pull request.

After you're happy with the proposed changes, you can merge the pull request. If you're working in a shared repository model, the proposed changes will be merged from the head branch to the base branch that was specified in the pull request. 

#### Upstream: When talking about a branch or a fork, the primary branch on the original repository is often referred to as the "upstream", since that is the main place that other changes will come in from. The branch/fork you are working on is then called the "downstream".

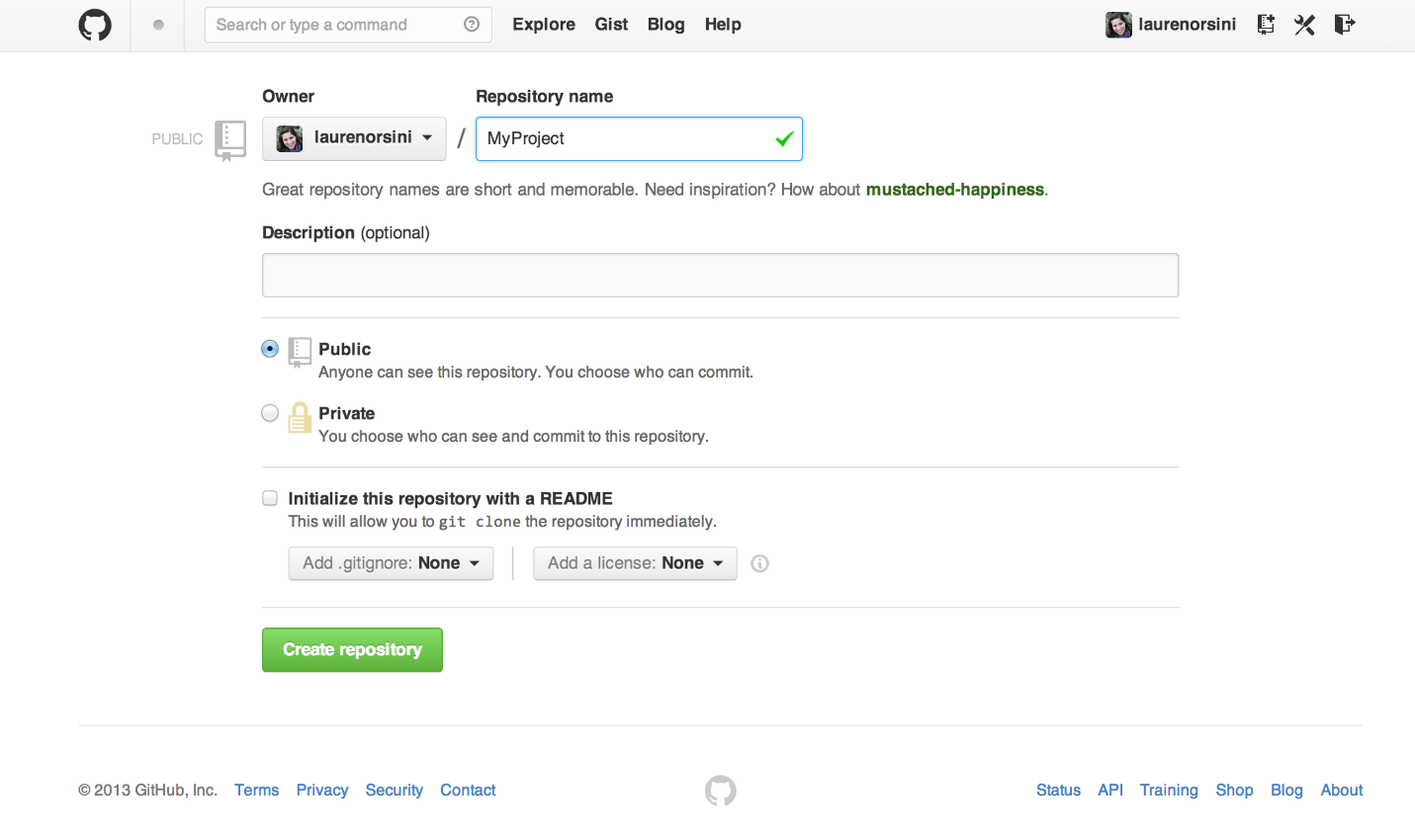
**Organization:** An Organization is exactly what it says. It’s a grouping of Projects and the Collaborators that work on the Organization’s projects. GitHub allows you to manage teams, projects and repositories within the Organization, making it easier for the members of the Owners team to manage permissions. So, if you take the hierarchy in a GitHub Organization then you would have something like this:



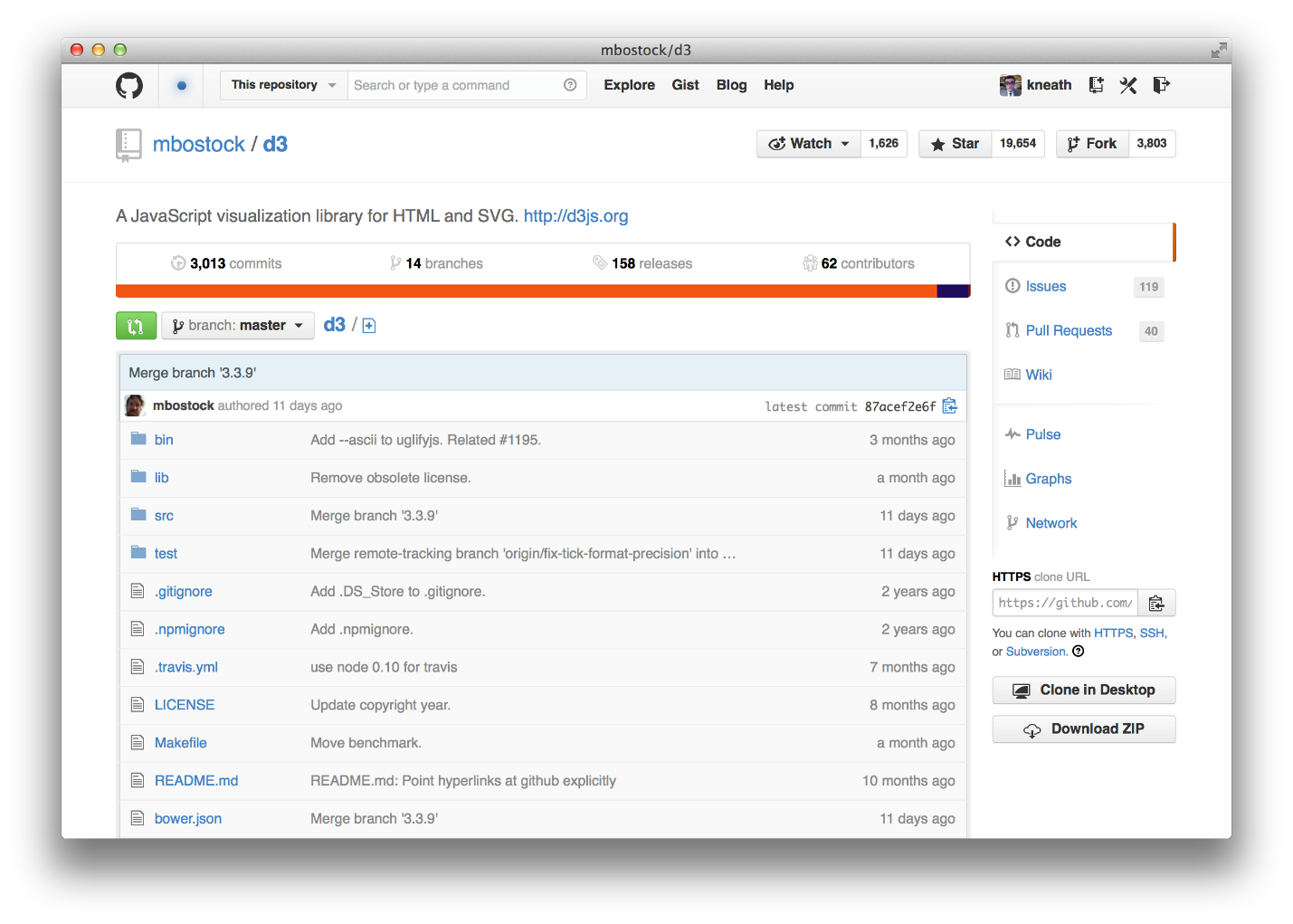
What we can see from above:

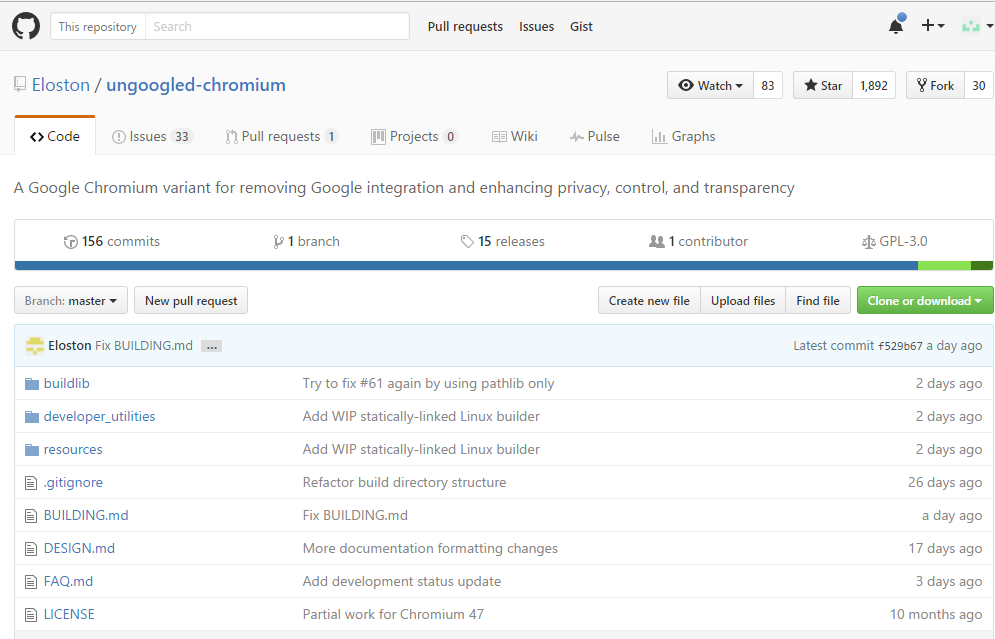
* Multiple collaborators working within an organization. They also have their own repositories.
* Specific repositories have been assigned to specific members

**Creating a Repository**



Once a repository is created, the user can push changes to the remote repository on GitHub.





Commit and changes

Number of pull requests

Issues

Description

Number of forks

Number of stars (likes)

Number of watchers

User handle

Name of repo

A Project (e.g. package managers) has multiple repos

A repo is a repository – a directory of code. E.g. Vundle.vim

A repo is owned by a unique user or company

A company has a page: <https://github.com/facebook>

* A company can have many repos e.g. “flow”
* A company has many “people” listed on Github. These people are “collaborators” in the dataset.

What does a repo contain:

* Code
* Commits – a commit contains some incremental code – additions and deletions
* First commit only has additions
* A commit is made by a unique user
* Pull request

There are two ways to make a change to a repository

* Pull request
* Collaborator pushes a change

A user can

* own one or more repositories.
* Have multiple followers (people)
* Follow multiple people
* A user can “star-gaze” multiple repositories – it is like following a repository

User activity

* Number of contributions by date
  + # of commits in different repositories
* Opened pull requests – i.e. user A can request user B to merge user A’s code into user B’s repository
* User A can merge a pull request from user B into user A’s repository
* User can open issues – highlight errors or issues in code that others can look at

Activity (contribution) is also called Event in the dataset

* E.g.

|  |  |
| --- | --- |
| **Front end name** | **Back end name** |
| Activity (contribution) | Event |
|  |  |
|  |  |
|  |  |

A user has

User A and user B.

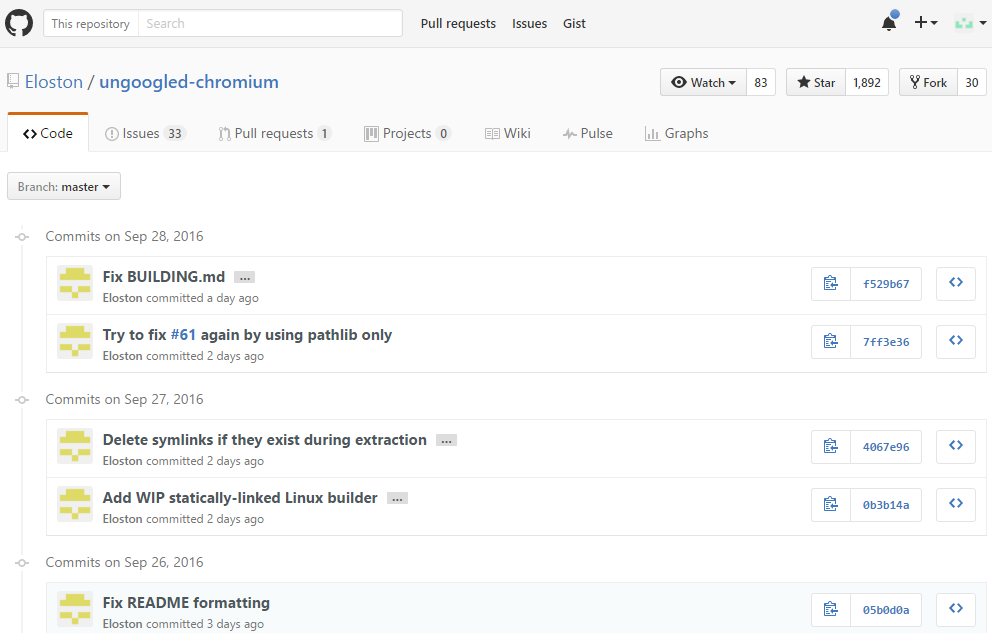
User A owns repo R.

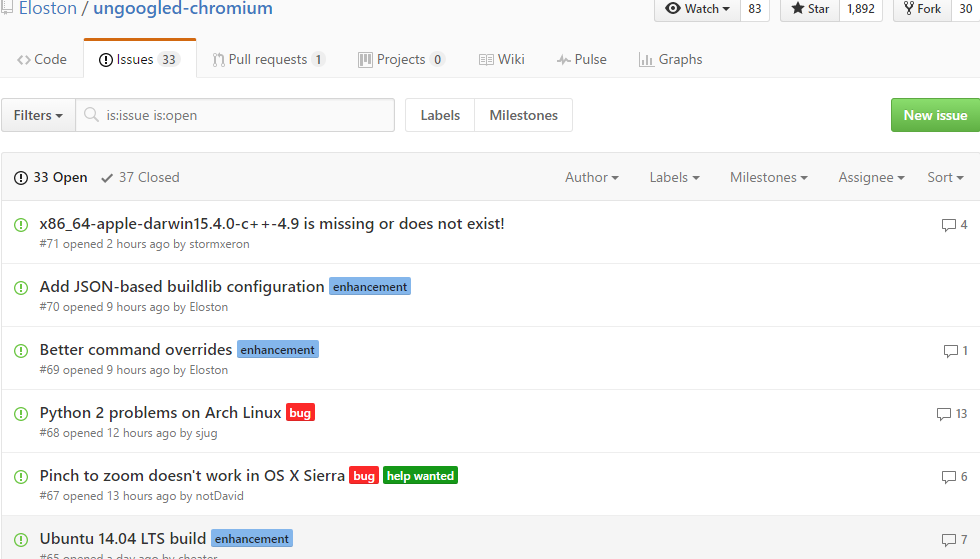
User A can commit to repo R.

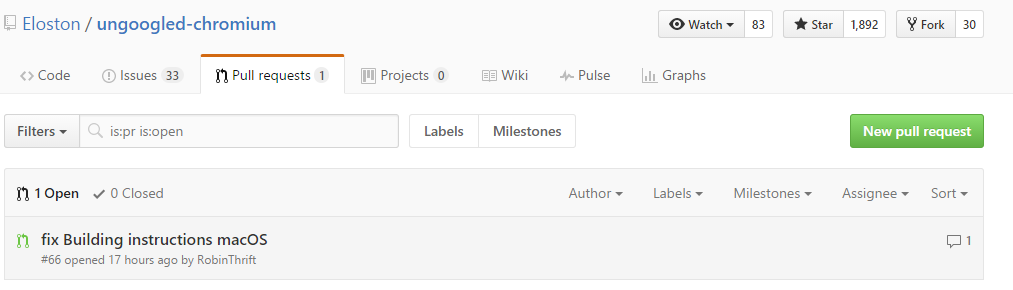
User B can commit to repo R only if B is a contributor on repo R.

**Understanding a Repository in detail**

* **Commits to the repository:** As mentioned before, commits are like snapshot of the code/text at a specific point in time. GitHub maintains this in a timeline



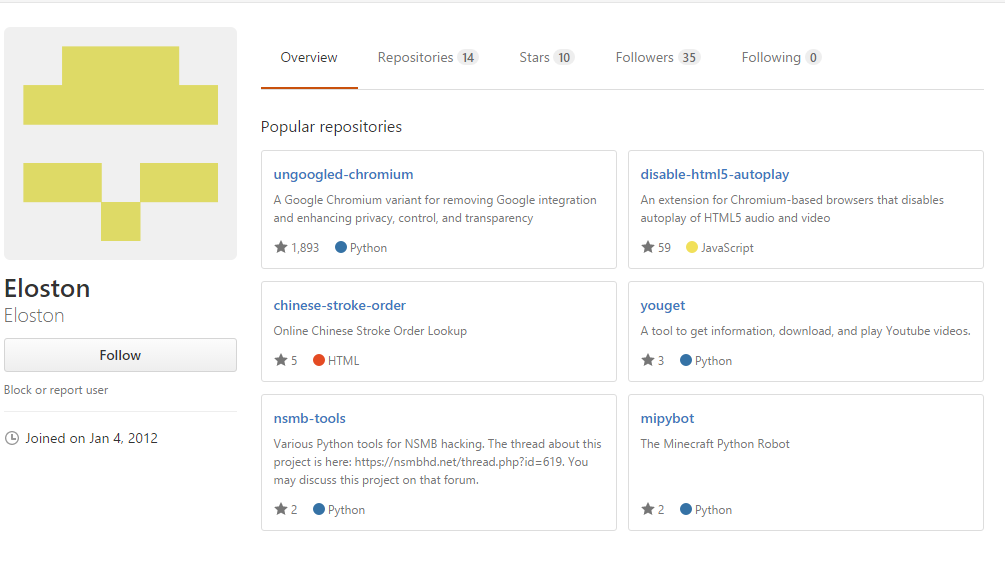
* **Open Issues**: Number of currently open issues
* 
* **Pull Requests**



* **Watchers**: Basically these many people are following this repository for any changes made. When any change is made to this repo (new commit or merge), a notification to these users will be sent.
* **Star** – This is just like “LIKE” in Facebook.
* **Fork** – Anyone can fork a publicly available repository and make his/her changes and publish/ or start a pull request for the changes to be merged.

**Understanding a user on GitHub**

A user is always public, irrespective whether s/he has private or public repositories.

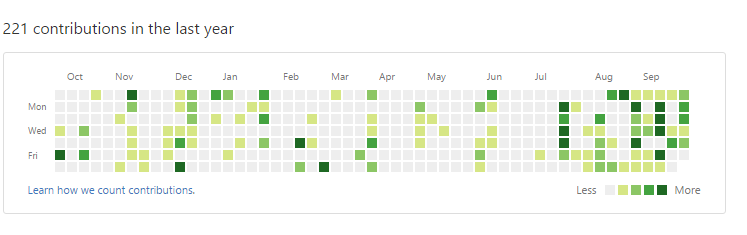


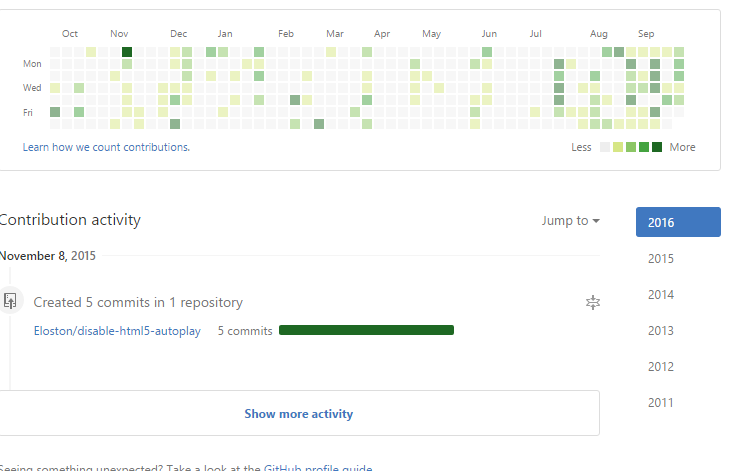
Repositories of the user

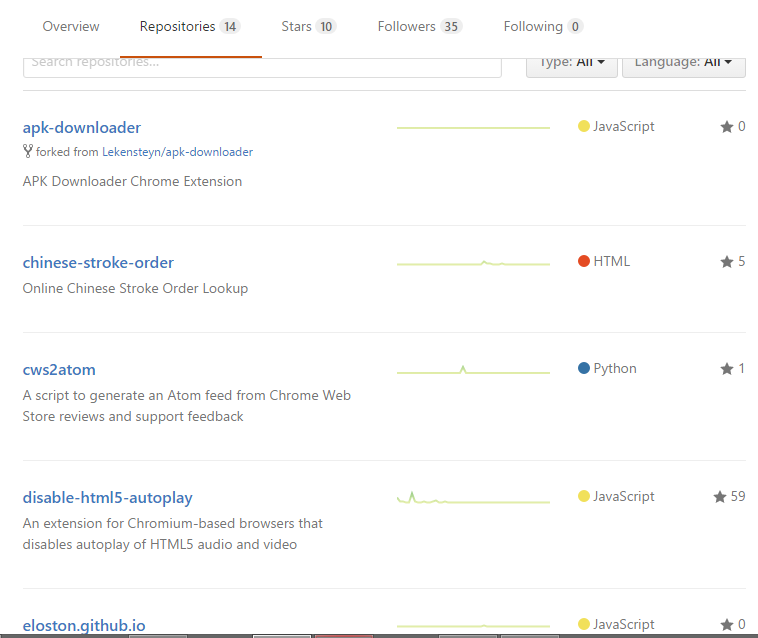
Follow a user

username

**Contributions by a user:** Any contribution by a distinct user can be defined as any activity – commit, comment, pull request done. This activity is tracked by the GitHub and is presented as a heatmap on the user’s timeline. Detail account can for a particular day can be found by clicking on that day. This only gives detail for public repositories and not the private ones.



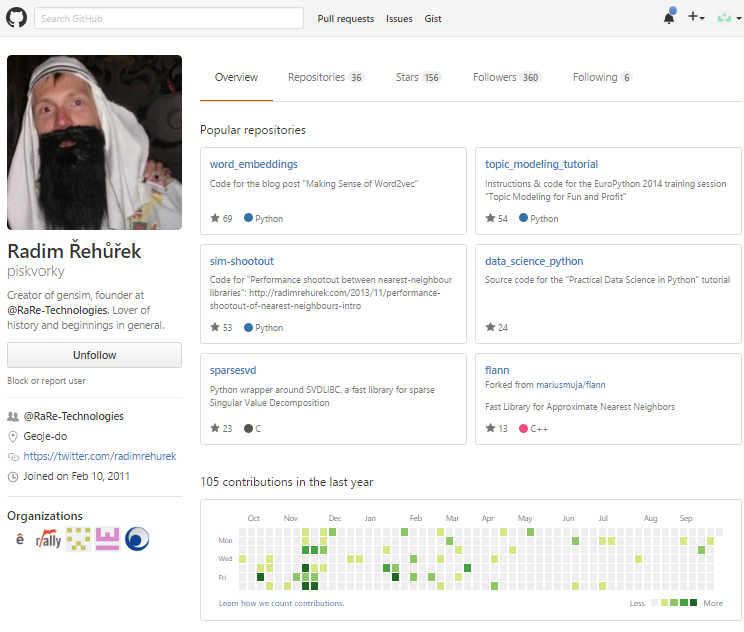




On the user’s page, we can see the following things:

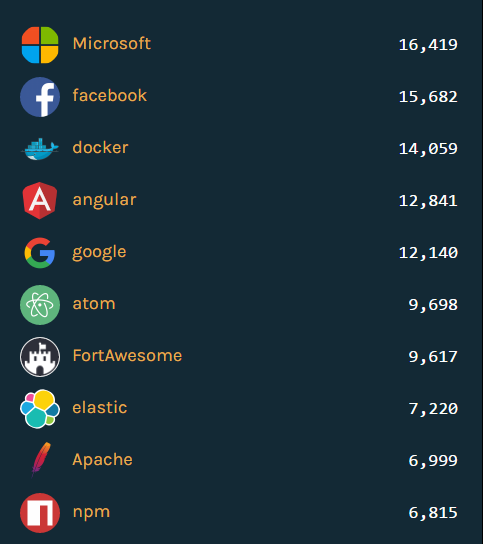
1. Repositories either forked or committed
2. Programming Language used
3. How many users have starred this repository
4. Overview contains the most common repositories by this user
5. Stars – this means how many stars this person has done
6. Followers – this means how many followers this user has
7. Following – this means how many users this user is following

A user’s page also has the organizations for which this user has contributed on GitHub.



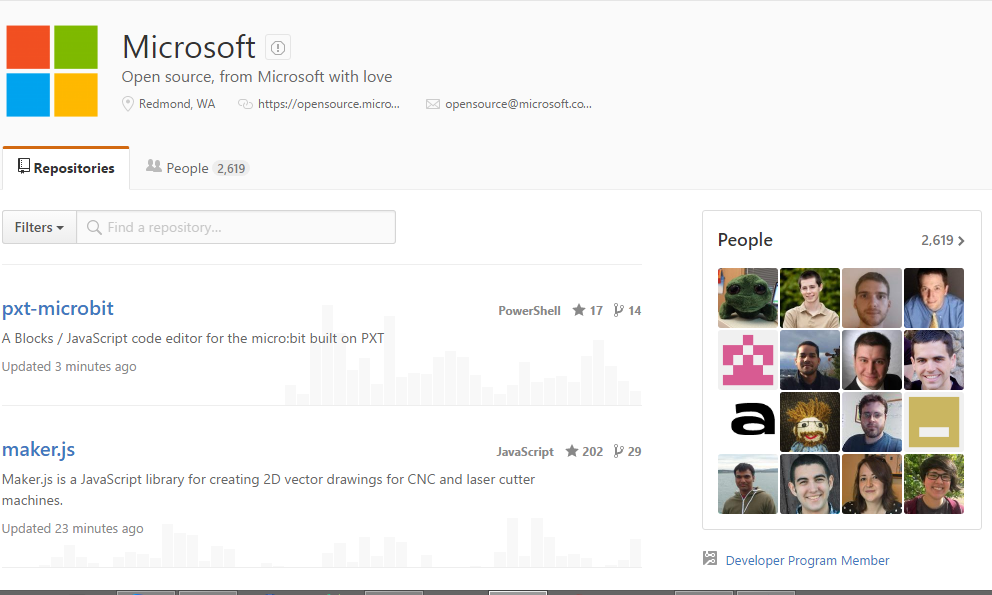
**Organization**

Many companies open source their code on GitHub. These include big names such as Google, Facebook, Microsoft.



Some sample pages of organization are below.



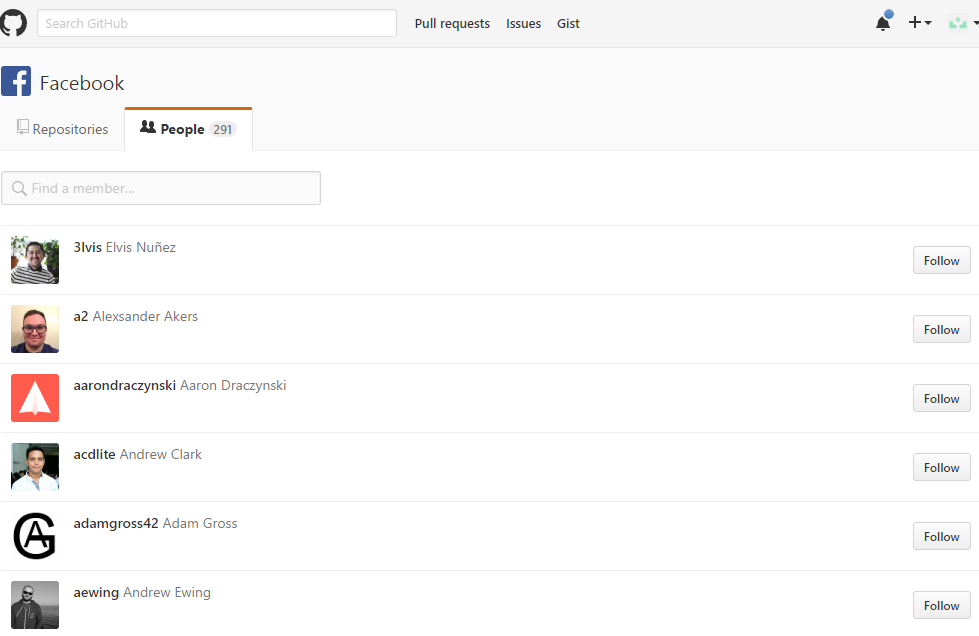


Each organization’s homepage will have several repositories listed





Each organization has a list of people who are either contributing to one of the repositories. These users might also have their own repositories on GitHub.



**Accessing GitHub Data**

Now that we know more about GitHub, we need to fetch data to understand more. To fetch more data we have multiple options:

1. GitHub API – The current version of the GitHub API is v3 and is a stable version. All requests must be made via a json and the output is also a json. There is a module in Python – PyGitHub. This works using version 3 and makes work easier.
   1. <https://github.com/PyGithub/PyGithub>
   2. Some activities require authentication, some don’t.
2. GitHub Archive - <https://www.githubarchive.org/>
   1. Contains all the past data.
   2. GitHub Archive is a project to record the public GitHub timeline, archive it, and make it easily accessible for further analysis.
   3. Activity archives are available starting 2/12/2011.
   4. Activity archives for dates between 2/12/2011-12/31/2014 was recorded from the (now deprecated) Timeline API.
   5. Activity archives for dates starting 1/1/2015 is recorded from the Events API.
   6. Structure of data: All entries are json format and the number of data points depend upon the event.
   7. {"id":"2489582592","type":"PushEvent","actor":{"id":1121789,"login":"Saisi","gravatar\_id":"","url":"https://api.github.com/users/Saisi","avatar\_url":"https://avatars.githubusercontent.com/u/1121789?"},"repo":{"id":25811730,"name":"Saisi/secret-octo-wookie","url":"https://api.github.com/repos/Saisi/secret-octo-wookie"},"payload":{"push\_id":536834835,"size":1,"distinct\_size":1,"ref":"refs/heads/master","head":"d864e94612d50e32d399d6514fd45d3776c0d594","before":"00d7ae0c0ca04f13cdee0eda271f27d41b27dca8","commits":[{"sha":"d864e94612d50e32d399d6514fd45d3776c0d594","author":{"email":"5070ee02c118bb3c67d539b31e44522403b2c763@users.noreply.github.com","name":"saisi"},"message":"1420113598 hmm","distinct":true,"url":"https://api.github.com/repos/Saisi/secret-octo-wookie/commits/d864e94612d50e32d399d6514fd45d3776c0d594"}]},"public":true,"created\_at":"2015-01-01T12:00:01Z"}
   8. {"id":"2489582599","type":"CreateEvent","actor":{"id":1400427,"login":"wizardbyron","gravatar\_id":"","url":"https://api.github.com/users/wizardbyron","avatar\_url":"https://avatars.githubusercontent.com/u/1400427?"},"repo":{"id":28685371,"name":"shiwuzhuquan/shiwuzhuquan.github.io","url":"https://api.github.com/repos/shiwuzhuquan/shiwuzhuquan.github.io"},"payload":{"ref":"master","ref\_type":"branch","master\_branch":"master","description":"","pusher\_type":"user"},"public":true,"created\_at":"2015-01-01T12:00:02Z","org":{"id":10363661,"login":"shiwuzhuquan","gravatar\_id":"","url":"https://api.github.com/orgs/shiwuzhuquan","avatar\_url":"https://avatars.githubusercontent.com/u/10363661?"}}
   9. Every event has the following attributes:
      1. Id of that event. This is unique across all the events
      2. Id of the user
      3. Login or username of the user
      4. Repository id and name to which this event was related to
      5. If there was some message/comment related with any particular event, what was it.
3. Google Big Query
   1. The entire GitHub Archive is also available as a public dataset on Google BigQuery: the dataset is automatically updated every hour and enables us to run arbitrary SQL-like queries over the entire dataset in seconds.
   2. The only issue with this is the dataset is massive and Google only provides 1 TB of free processed data per month. Moreover, without a subscription, the options are very less.
   3. The entire GitHub Archive is also available as a public dataset on [Google BigQuery](https://developers.google.com/bigquery/): the dataset is automatically updated every hour and enables you to run [arbitrary SQL-like queries](https://developers.google.com/bigquery/docs/query-reference) over the entire dataset in seconds. To get started:
      1. If you don't already have a Google project...
      2. [Login into the Google Developer Console](https://console.developers.google.com/)
      3. [Create a project](https://developers.google.com/console/help/#creatingdeletingprojects) and [activate the BigQuery API](https://developers.google.com/console/help/#activatingapis)
      4. Open public dataset: <https://bigquery.cloud.google.com/table/githubarchive:day.20150101>
      5. Execute your first query...
         1. /\* count of issues opened, closed, and reopened between 1/1/2015 and 2/1/2015 \*/   
              
            SELECT event as issue\_status, COUNT(\*) as cnt FROM ( SELECT type, repo.name, actor.login, JSON\_EXTRACT(payload, '$.action') as event, FROM (TABLE\_DATE\_RANGE([githubarchive:day.], TIMESTAMP('2015-01-01'), TIMESTAMP('2015-02-01') )) WHERE type = 'IssuesEvent' ) GROUP by issue\_status;
      6. Schema

|  |  |  |  |
| --- | --- | --- | --- |
| **type** | STRING | NULLABLE | https://developer.github.com/v3/activity/events/types/ |
| **public** | BOOLEAN | NULLABLE | Always true for this dataset since only public activity is recorded. |
| **payload** | STRING | NULLABLE | Event payload in JSON format |
| **repo** | RECORD | NULLABLE | Repository associated with the event |
| **repo.id** | INTEGER | NULLABLE | Numeric ID of the GitHub repository |
| **repo.name** | STRING | NULLABLE | Repository name |
| **repo.url** | STRING | NULLABLE | Repository URL |
| **actor** | RECORD | NULLABLE | Actor generating the event |
| **actor.id** | INTEGER | NULLABLE | Numeric ID of the GitHub actor |
| **actor.login** | STRING | NULLABLE | Actor's GitHub login |
| **actor.gravatar\_id** | STRING | NULLABLE | Actor's Gravatar ID |
| **actor.avatar\_url** | STRING | NULLABLE | Actor's Gravatar URL |
| **actor.url** | STRING | NULLABLE | Actor's profile URL |
| **org** | RECORD | NULLABLE | GitHub org of the associated repo |
| **org.id** | INTEGER | NULLABLE | Numeric ID of the GitHub org |
| **org.login** | STRING | NULLABLE | Org's GitHub login |
| **org.gravatar\_id** | STRING | NULLABLE | Org's Gravatar ID |
| **org.avatar\_url** | STRING | NULLABLE | Org's Gravatar URL |
| **org.url** | STRING | NULLABLE | Org's profile URL |
| **created\_at** | TIMESTAMP | NULLABLE | Timestamp of associated event |
| **id** | STRING | NULLABLE | Unique event ID |
| **other** | STRING | NULLABLE | Unknown fields in JSON format |



1. GhTorrent
   1. GHTorrent monitors the Github public event time line. For each event, it retrieves its contents and their dependencies, exhaustively. It then stores the raw JSON responses to a MongoDB database, while also extracting their structure in a MySQL database.
   2. MongoDB Dump
      1. Collections in MongoDB
      2. Here is a list of collections along with the Github API URL data is cached data. All URLs need to be prefixed with https://api.github.com/. In MongoDB, each entity is by default indexed by the parameter fields in each corresponding URL.

| **Collection name** | **Github API URL** | **Documentation URL** |
| --- | --- | --- |
| commit\_comments | #{user}/#{repo}/commits/#{sha}/comments | [commit comments](http://developer.github.com/v3/repos/comments/#list-comments-for-a-single-commit) |
| commits | repos/#{user}/#{repo}/commits | [commits](http://developer.github.com/v3/repos/commits/#list-commits-on-a-repository) |
| events | events | [events](http://developer.github.com/v3/activity/events/) |
| followers | users/#{user}/followers | [followers list](http://developer.github.com/v3/users/followers/#list-followers-of-a-user) |
| forks | repos/#{user}/#{repo}/forks | [forks list](http://developer.github.com/v3/repos/forks/#list-forks) |
| issues | /repos/#{owner}/#{repo}/issues | [issues for a repo](http://developer.github.com/v3/issues/#list-issues-for-a-repository) |
| issue\_comments | repos/#{owner}/#{repo}/issues/comments/#{comment\_id} | [issue comments](http://developer.github.com/v3/issues/comments/#list-comments-on-an-issue) |
| issue\_events | repos/#{owner}/#{repo}/issues/events/#{event\_id} | [issue events](http://developer.github.com/v3/issues/events/) |
| org\_members | orgs/#{org}/members | [organization members](http://developer.github.com/v3/orgs/members/) |
| pull\_request\_comments | repos/#{owner}/#{repo}/pulls/#{pullreq\_id}/comments | [pull request review comments](http://developer.github.com/v3/pulls/comments/%22) |
| pull\_requests | repos/#{user}/#{repo}/pulls | [pull requests](http://developer.github.com/v3/pulls/) |
| repo\_collaborators | repos/#{user}/#{repo}/collaborators | [repo collaborators](http://developer.github.com/v3/repos/collaborators/) |
| repo\_labels | repos/#{owner}/#{repo}/issues/#{issue\_id}/labels | [issue labels](http://developer.github.com/v3/issues/labels/#list-all-labels-for-this-repository) |
| repos | repos/#{user}/#{repo} | [repositories](http://developer.github.com/v3/repos/#list-all-public-repositories) |
| users | users/#{user} | [users](http://developer.github.com/v3/users/#get-a-single-user) |
| watchers | repos/#{user}/#{repo}/stargazers | [stargazers](http://developer.github.com/v3/activity/starring/#list-stargazers) |

* + 1. How much data GhTorrent has?
       1. Currently (Jan 2015), MongoDB stores around 4TB of JSON data (compressed), while MySQL more than 1.5 billion rows of extracted metadata. A large part of the activity of 2012, 2013, 2014 and 2015 has been retrieved, while we are also going backwards to retrieve the full recorded history of important projects.
       2. Data Structure
          1. Commit Comments

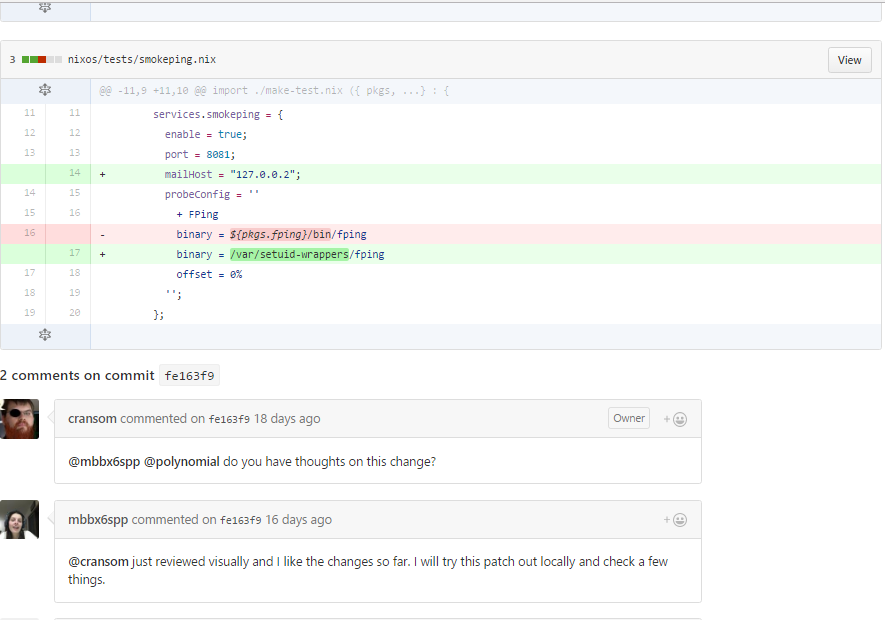
{ "\_id" : ObjectId("57e5e0816480fda9b7bf56c6"), "url" : "https://api.github.com/repos/cransom/nixpkgs/comments/19155649", "html\_url" : "https://github.com/cransom/nixpkgs/commit/fe163f9c2fd4d7d59964a8c4d745b435b3023d27#commitcomment-19155649", "id" : 19155649, "user" : { "login" : "cransom", "id" : 1957293, "avatar\_url" : "https://avatars.githubusercontent.com/u/1957293?v=3", "gravatar\_id" : "", "url" : "https://api.github.com/users/cransom", "html\_url" : "https://github.com/cransom", "followers\_url" : "https://api.github.com/users/cransom/followers", "following\_url" : "https://api.github.com/users/cransom/following{/other\_user}", "gists\_url" : "https://api.github.com/users/cransom/gists{/gist\_id}", "starred\_url" : "https://api.github.com/users/cransom/starred{/owner}{/repo}", "subscriptions\_url" : "https://api.github.com/users/cransom/subscriptions", "organizations\_url" : "https://api.github.com/users/cransom/orgs", "repos\_url" : "https://api.github.com/users/cransom/repos", "events\_url" : "https://api.github.com/users/cransom/events{/privacy}", "received\_events\_url" : "https://api.github.com/users/cransom/received\_events", "type" : "User", "site\_admin" : false }, "position" : null, "line" : null, "path" : null, "commit\_id" : "fe163f9c2fd4d7d59964a8c4d745b435b3023d27", "created\_at" : "2016-09-24T02:10:04Z", "updated\_at" : "2016-09-24T02:10:04Z", "body" : "@mbbx6spp @polynomial do you have thoughts on this change?" }

A mongodb dump from GHTorrent would be used to analyze data in the first step. The steps to install MongoDB on a windows machine

* Go to this link on [Mongodb’s website](https://www.mongodb.com/download-center#community) to download.
* Once the download is finished, double click the MSI file to install with default settings.
* You need to start mongoDB database with a command after you create a data directory. This directory can be anywhere. I create this directory in the C:\ drive with the name – mongodata
* Once you are done with above, just go to bin directory inside the installation folder. It should be
* cd C:\mongodb\bin\
* then run
* mongod --dbpath=C:/mongodata/
* All this should get the db started.
* You can use a GUI tool to run the commands without having to visit the command line to run all those queries.
* I use [RoboMongo](https://robomongo.org/) and its fine for my use. Moreover, most of the times I interact with the db using Python and I use this GUI to look at the data structure or whether my operations are going through.

Collections:

1. Commit\_Comments: When a commit is made by a user on his/her repo, different users can comment on this commit. Mostly, its done when let’s say an user A finds an issue with the code and asks the owner O of the repository to fix the error. Once O commits a new code after fixing the potential issue, users can interact on this page.  
     
   This collection specifically caters to the comments made by users and does not contain any information on the commit itself. There is another collection for that.  
     
   Sample URL : <https://github.com/cransom/nixpkgs/commit/fe163f9c2fd4d7d59964a8c4d745b435b3023d27#commitcomment-19155649>  
     
   Snapshot



Comments on the commit made by different users related to this commit

Commit changes

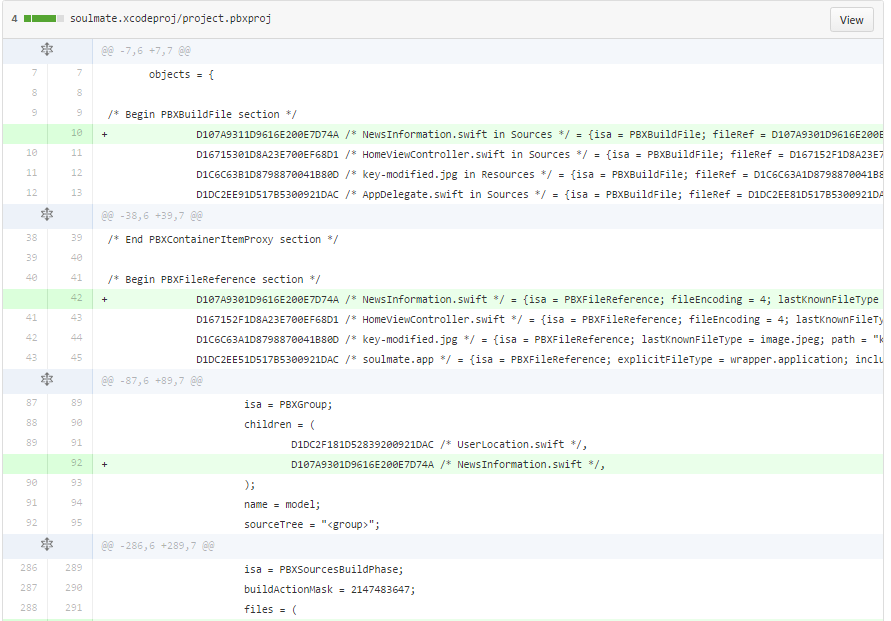
Green is addition

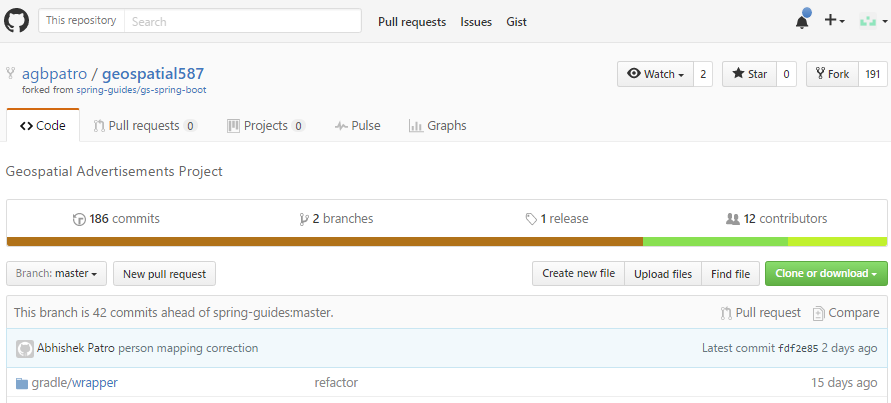
Red is deletion

1. Commits: This collection contains all the data regarding every single commit made by users. The fields contained in this are:
   1. Sha of the commit
   2. url of the commit
   3. author information of the commit
   4. number of additions, deletions on the whole for this commit
   5. number of additions, deletions, on each file that has been changes

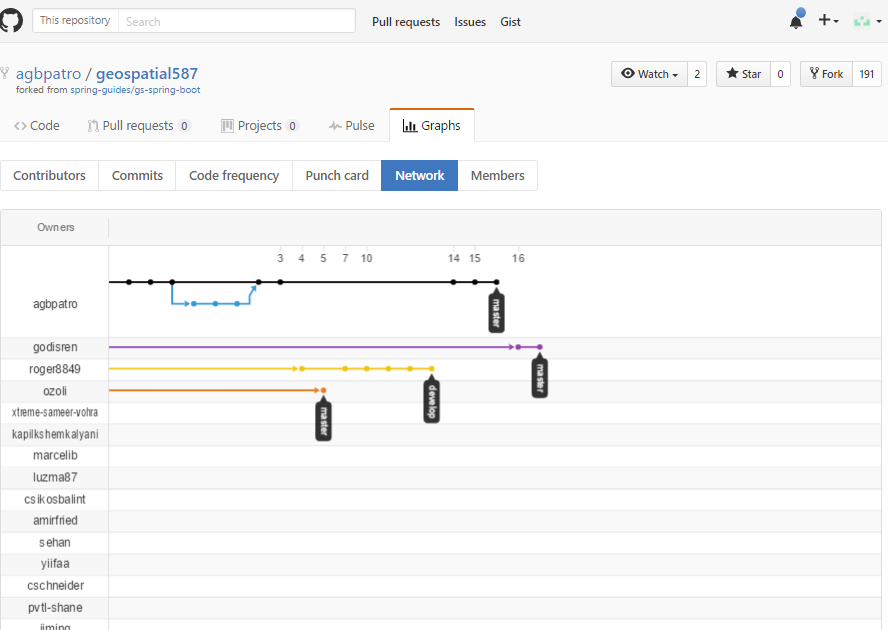
This collection contains all commits by the user.  
  
Sample url : <https://github.com/rhsu0268/soulmate/commit/93c3f4e7840ceeeed8ed647b8a5bb253369dfee6>

Snapshot:

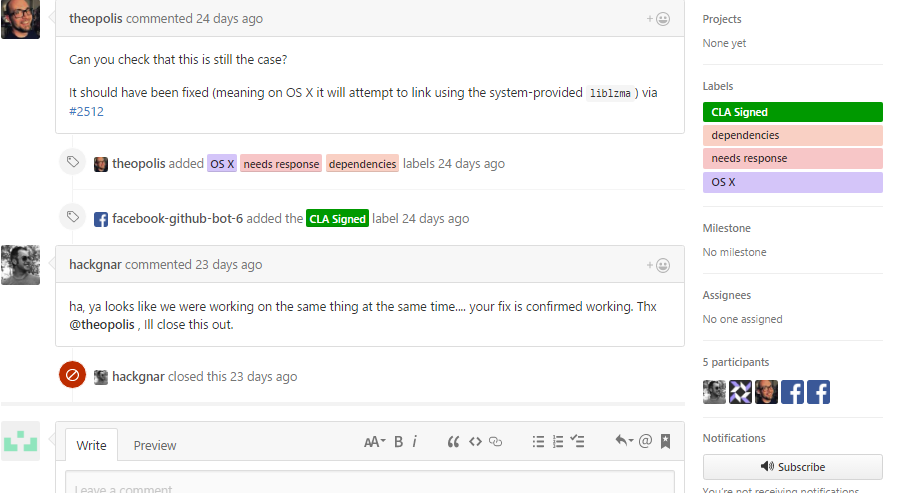


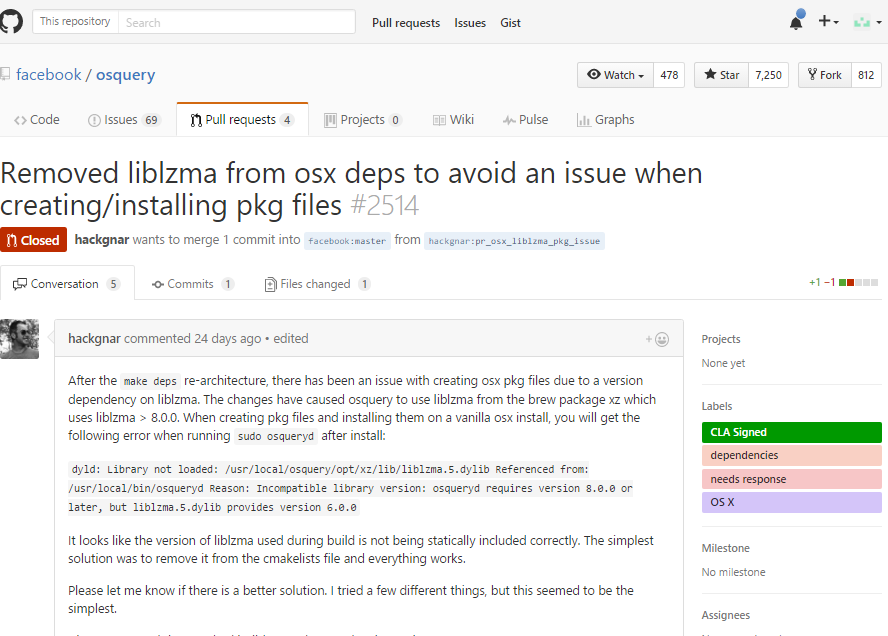
1. forks: This event is created every time a repository is forked by any user. One thing that I noticed is although, a key fork is created with value true but forks\_count and forks have value 0 although there are forks visible on the website. This is at a repository level.  
     
   

Fork event

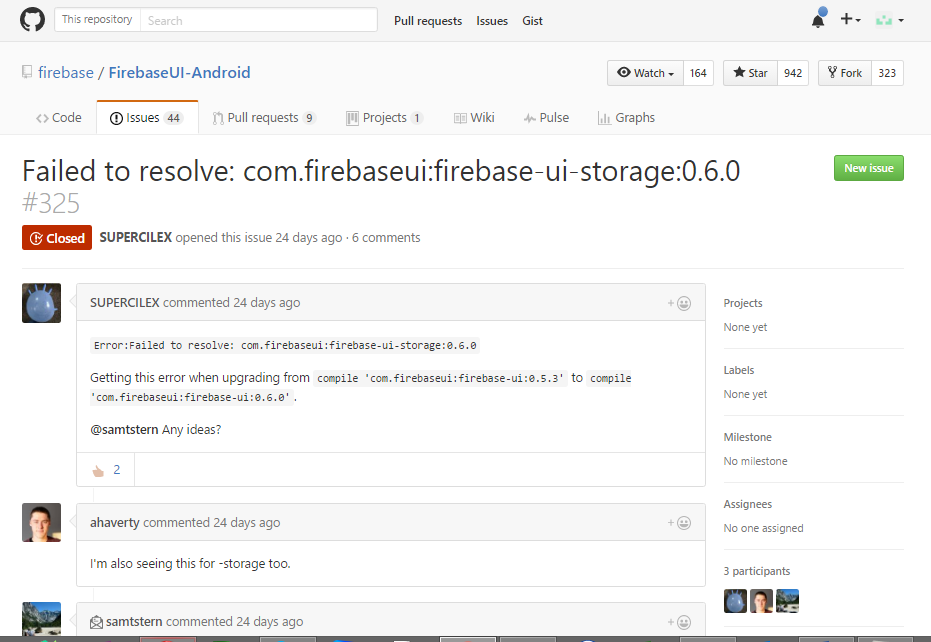


1. Issue\_comments: Any user can comment on any open issue on any repository. This event is created every time a comment is made on an issue on the repository. This has the following important fields:
   1. url of the repository
   2. name of the user who commented
   3. url of the issue
   4. what was the comment  
      Example: https://github.com/facebook/osquery/pull/2514#issuecomment-249338903





1. Issue\_events: This has information about users related to event such as issue being closed or assigned.
2. Issues: This has information about all issues related to a repository. This is at an issue level, so a user can create more than one issues on a repository. The important parts are:
   1. User information
   2. Title of issue
   3. Body of issue
   4. Creation and updatation time
   5. Whether issue is still open/closed
   6. To whom the issue is assigned  
        
        
      Example url : https://github.com/firebase/FirebaseUI-Android/issues/325

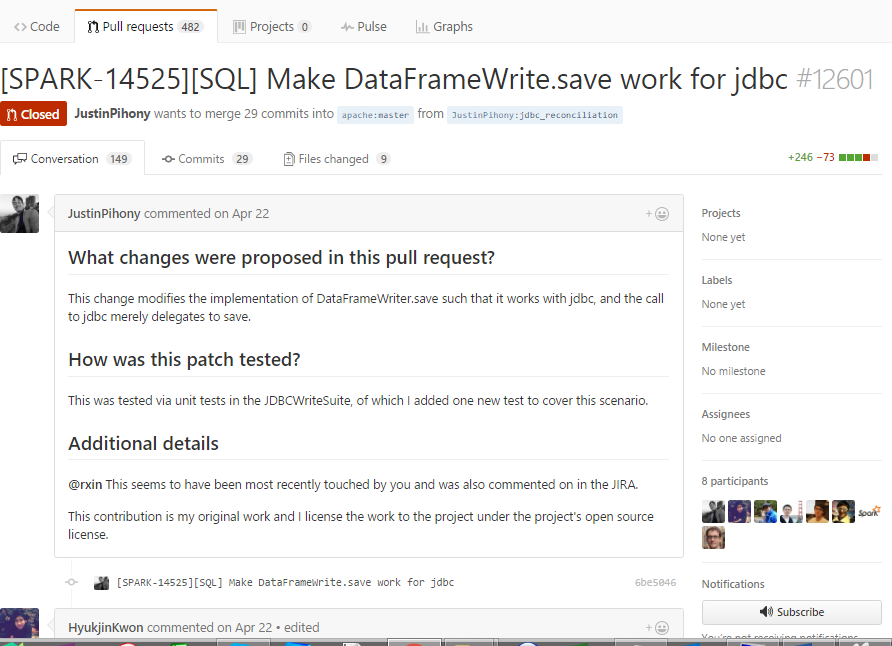


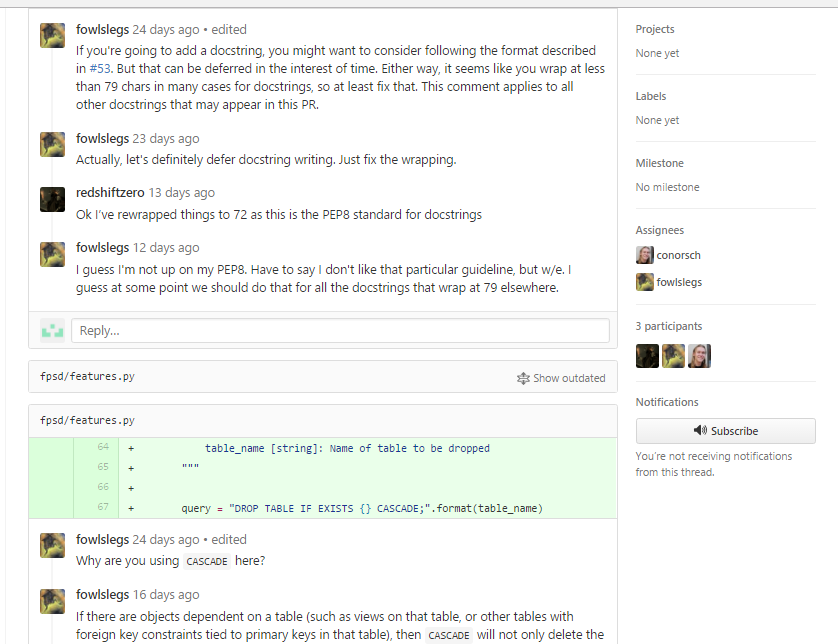
Further comments

Body of issue

title

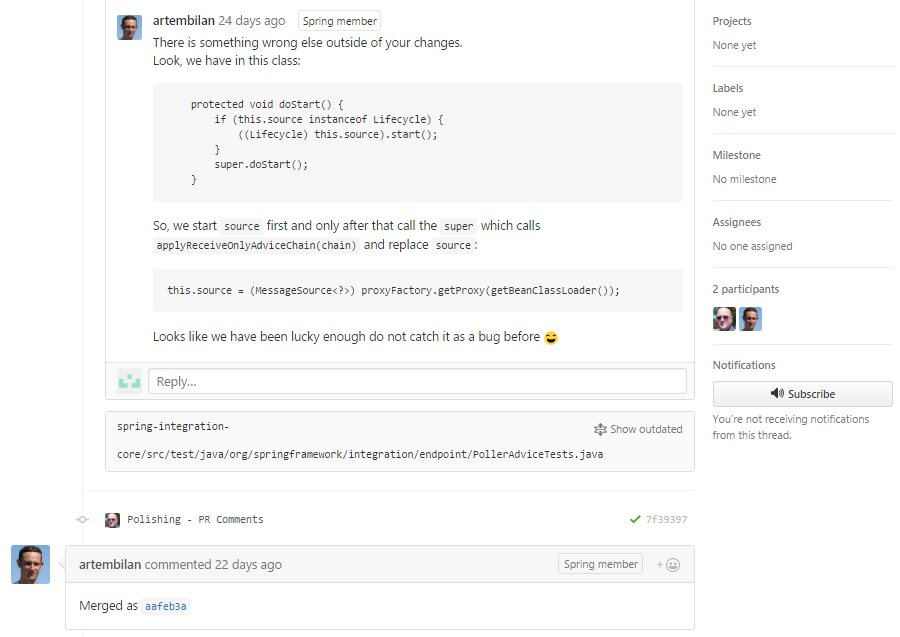
1. Org\_members: contains discussion on the pull requests by different users.





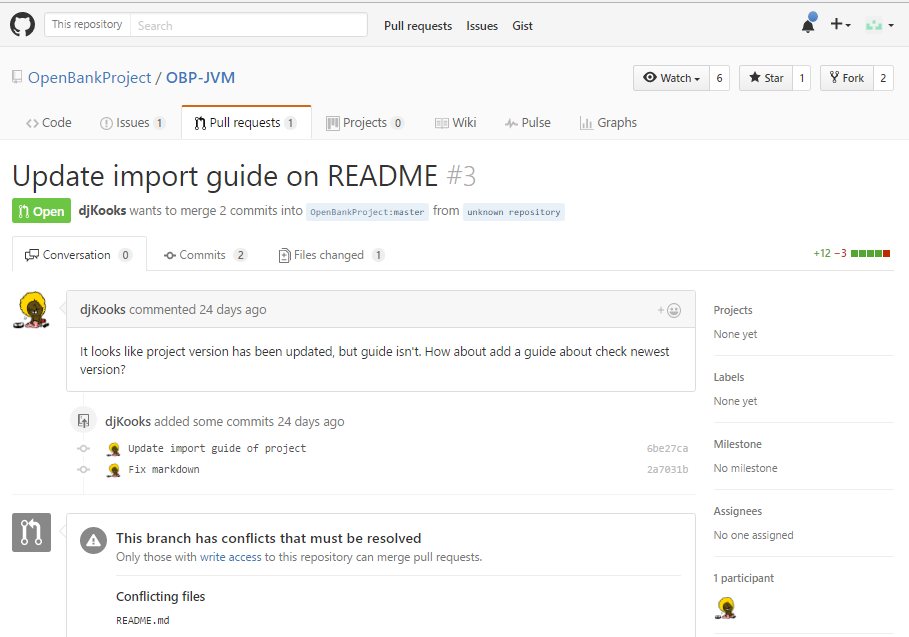
This is mostly regarding the changes that has been proposed by a user

1. Pull\_request\_comment: same as above

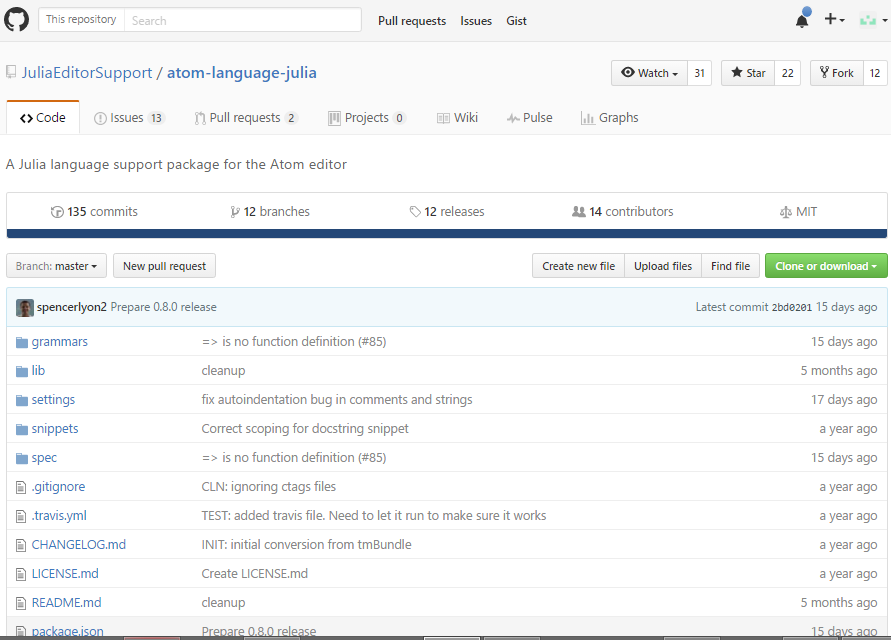


1. Pull Requests: When any pull request is made by any use on any repository. This can be related to simple changes in the code to major changes to adding documentation or demo codes.

Sample url : https://github.com/OpenBankProject/OBP-JVM/pull/3



1. Repos: contains all information about a repository on GitHub. Contains information on:
   1. Owner
   2. Html url
   3. Forks
   4. Star gazer count



1. Users: contains basic information on all users



1. Star gazers: contains basic information on star gazers

**Find 1000 unique users in the commits collections**

* 1. **Find User level data from the commits collection**

1. Create an empty collection named unique\_users
2. Run the following query

db.getCollection('users').distinct("login").forEach(function(usr){

db.unique\_users.insert({'user': usr});

});

1. The query above inserts all information on unique users in this new table
   1. Finding user level data from commit\_comments collection

db.getCollection('commit\_comments').distinct("user.login")

* 1. Finding user level data from events collection

db.getCollection('events’).distinct("actors.login")

* 1. Issue\_comments

db.getCollection(‘issue\_comments’).distinct("users.login")

* 1. Issue\_events

db.getCollection('issue\_events’).distinct("actors.login")

* 1. Issues

db.getCollection('issues’).distinct("user.login")

* 1. Pull request comments

db.getCollection(pull\_requests).distinct("user.login")

* 1. Repos

db.getCollection(repos).distinct("owner.login")

* 1. Users

db.getCollection(users).distinct("login")

* 1. Watchers

db.getCollection(watchers).distinct(“login")

1. Join collections to get user activity
   1. User and user commit comments

db.unique\_users.aggregate([{

$lookup: {

from: "commit\_comments",

localField: "login",

foreignField: "user.login",

as: "user\_commit\_comments"

}

}])

* 1. User and events

**GITHUB MYSQL Database (via GHTORRENT)**

## **Entities and their relationships**

#### users

Github users.

* A user has a unique user name or email. May contain artificially generated user names, see commits below.
* There are two types of users, USERs and ORGanizations.
  + Users can be real or fake. Real users can own projects and perform actions such as open issues, create pull requests and push commits. Fake users only appear as authors or committers of commits. Fake users are marked by the fake field.
  + Organizations are meta users that point to a collection of users. The members of organizations can be found in organization\_members. Organization users can only own projects and they do not perform any other actions.
* Users may be marked as deleted. This means that the user was once active on GitHub but GHTorrent can no longer get his/her details.

Update Nov 2015: User entries are now geocoded. The location field remains intact, while 5 fields have been added with information about the geographic location of the user. The Open Street Maps API has been used to do the mapping of the location field to the user's geocode. As a result, the state and city fields are stored in the local language of the geocoded area. Also, many users do not report their location or their location is field in with random information; in those cases, no geocoding information is available.

**organization\_members**

Users that are members of an organization.

* The created\_at field is only filled in accurately for memberships for which GHTorrent has recorded a corresponding event. Otherwise, it is filled in with the latest date that the corresponding user or organization has been created.

*Update Nov 2015:* Organizations can now select wheather membership information is revealed to external parties. This means that information about this table can no longer be accurate.

**projects**

Information about repositories. A repository is always owned by a user.

* The forked\_from field is empty unless the project is a fork in which case it contains the id of the project the project is forked from.
* The deleted field means that the project has been deleted from Github.
* The updated\_at field indicates when the last full update was done for this project.

**project\_members**

Users that have commit access to the repository.

The created\_at field is only filled in accurately for memberships for which GHTorrent has recorded a corresponding event. Otherwise, it is filled in with the latest date that the corresponding user or project has been created.

### **project\_languages**

Languages that are used in the repository along with **byte counts** for all files in those languages.

Multiple entries can exist per project. The created\_at field is filled in with the latest timestamp the query for a specific project\_id was done.

The table is filled in when the project has been first inserted on when an update round for all projects is made.

#### commits

Unique commits.

* Each commit is identified globally through its sha field. If the author or the committer has not configured his [Github email address](https://help.github.com/articles/setting-your-email-in-git), no resolution to a user entry is possible. In that case, GHTorrent generates artificial users using the provided email in the Git commit author or committer fields. If the user then configures his Github account, GHTorrent will update the artificial user accordingly.
* The project\_id field contains a link to the project that this commit has been first associated with. This might not be the project this commit was initially pushed to, e.g. in case the fork is processed before the parent. See [project\_commits](http://ghtorrent.org/relational.html#project_commits).
* The project\_id field may be null when the repository has been deleted at the time the commit is processed. This situation might happen when retrospectively processing pull requests for a repository and the repository which the pull request originates from has been deleted.

#### commit\_parents

The parent commit(s) for each commit, as specified by Git.

#### project\_commits

The commits belonging to the history of a project.

More than one projects can share the same commits if one is a fork of the other.

#### commit\_comments

Code review comments on commits.

These are comments on individual commits. If a commit is associated with a pull request, then its comments are in the [pull\_request\_comments](http://ghtorrent.org/relational.html#pull_request_comments) table.

#### followers

A follower to a user.

The created\_at field is only filled in accurately for followships for which GHTorrent has recorded a corresponding event. Otherwise, it is filled in with the latest date that the corresponding user or follower has been created.

#### watchers

Users that have starred (was [watched](https://github.com/blog/1204-notifications-stars)) a project

The created\_at field is only filled in accurately for starrings for which GHTorrent has recorded a corresponding event. Otherwise, it is filled in with the latest date that the corresponding user or project has been created.

#### pull\_requests

A pull request initiated from head\_repo\_id:head\_commit\_id to base\_repo\_id:base\_commit\_id

* Pull requests can be in various states. The states and their transitions are recorded in the [pull\_request\_history](http://ghtorrent.org/relational.html#pull_request_history) table.
* The pullreq\_id field is Github's pull request unique identifier
* The intra\_branch field signifies that the head and base repositories are the same
* If the head repository is NULL, this means that the corresponding project had been deleted when GHTorrent processed the pull request.

#### pull\_request\_history

An event in the pull request lifetime

The action field can take the following values

* opened: When the pull request has been opened
* closed: When the pull request has been closed
* merged: When Github detected that the pull request has been merged. No merges outside Github (i.e. Git based) are reported
* reoponed: When a pull request is opened after being closed
* syncrhonize: When new commits are added/removed to the head repository

#### pull\_request\_commits

A commit associated with a pull request

The list is additive. This means if a rebase with commit squashing takes place after the commits of a pull request have been processed, the old commits will not be deleted.

#### pull\_request\_comments

A code review comment on a commit associated with a pull request

The list is additive. If commits are squashed on the head repo, the comments remain intact.

#### issues

An issue associated with a repository

* The assignee field is filed in with the user to which the issue was assigned at the time the issue was processed.
* Issues have history recorded in the [issue\_events](http://ghtorrent.org/relational.html#issue_events) table.
* For every pull request, GHTorrent creates a corresponding issue. The pull\_request\_id field points to the associated pull request
* The issue\_id field is the unique identifier given to the issue by Github.

#### issue\_events

An event on an issue

* The action field can have the following values:
  + subscribed: When a user subscribes to receive notifications about the issue.
  + mentioned: When a user is mentioned by another user (@user notation)
  + closed: When the issue has been closed
  + referenced: The issue was referenced in a commit (using the [fixes: conventions](https://github.com/blog/831-issues-2-0-the-next-generation))
  + assigned: When the issue has been assigned to an actor.
  + reopened: When a closed issue is reopened
  + unsubscribed: When a user unsubscribed from issue.
  + merged: When the pull request pointed by the issue has been merged.
  + head\_ref\_cleaned: (Not documented) ?
  + head\_ref\_deleted: (Not documented) When the branch of the head repository has been deleted
  + head\_ref\_restored: (Not documented) When the head repository of a pull request has been restored (using the restore branch functionality).
* The action\_specific field gets filled in with the commit\\_id of the last commit when a pull request has been closed, merged or referenced.

#### issue\_comments

An entry to the issue discussion. This table is always filled in with pull request (or issue) discussion comments, irrespective of whether the repository has issues enabled or not.

#### repo\_labels

A label to be assigned to an issue affecting this repository.

#### issue\_labels

A label that has been assigned to an issue

## **Example queries**

#### List commits for a repository

**select** **c**.**\***

**from** commits **c**, project\_commits pc, projects p, users u

**where** u.login **=** 'rails'

**and** p.name **=** 'rails'

**and** p.id **=** pc.project\_id

**and** **c**.id **=** pc.commit\_id

**order** **by** **c**.created\_at **desc**

#### Get all actions for a pull request

**select** **user**, action, created\_at **from**

(

**select** prh.action **as** action, prh.created\_at **as** created\_at, u.login **as** **user**

**from** pull\_request\_history prh, users u

**where** prh.pull\_request\_id **=** **?**

**and** prh.actor\_id **=** u.id

**union**

**select** ie.action **as** action, ie.created\_at **as** created\_at, u.login **as** **user**

**from** issues i, issue\_events ie, users u

**where** ie.issue\_id **=** i.id

**and** i.pull\_request\_id **=** **?**

**and** ie.actor\_id **=** u.id

**union**

**select** 'discussed' **as** action, ic.created\_at **as** created\_at, u.login **as** **user**

**from** issues i, issue\_comments ic, users u

**where** ic.issue\_id **=** i.id

**and** u.id **=** ic.user\_id

**and** i.pull\_request\_id **=** **?**

**union**

**select** 'reviewed' **as** action, prc.created\_at **as** created\_at, u.login **as** **user**

**from** pull\_request\_comments prc, users u

**where** prc.user\_id **=** u.id

**and** prc.pull\_request\_id **=** **?**

) **as** actions

**order** **by** created\_at;

#### Get participants in an issue or pull request

**select** **distinct**(user\_id) **from**

(

**select** user\_id

**from** pull\_request\_comments

**where** pull\_request\_id **=** **?**

**union**

**select** user\_id

**from** issue\_comments ic, issues i

**where** i.id **=** ic.issue\_id **and** i.pull\_request\_id **=** **?**

) **as** participants

#### Get all users in NL that committed to a Java project today

**select** u.login

**from** users u, commits **c**, projects p, project\_commits pc

**where** date(**c**.created\_at) **=** date(now())

**and** pc.commit\_id **=** **c**.id

**and** **c**.author\_id **=** u.id

**and** u.country\_code **=** 'nl'

**and** 'java' **=** (**select** pl.**language**

**from** project\_langauges pl

**where** pl.project\_id **=** p.id

**order** **by** pl.created\_at **desc**, pl.bytes **desc**

**limit** 1)

*-- Get the latest byte count for languges in Ruby on Rails*

**select** **\***

**from** project\_languages

**where** project\_id **=** 1334

**order** **by** created\_at **desc**

*--- Get active core team participants for the last 3 months*

**select** **distinct**(u.login) **as** login

**from** commits **c**, users u, project\_commits pc, users u1, projects p

**where** u.id **=** **c**.committer\_id

**and** u.fake **is** **false**

**and** pc.commit\_id **=** **c**.id

**and** pc.project\_id **=** p.id

**and** p.owner\_id **=** u1.id

**and** p.name **=** 'rails'

**and** u1.login **=** 'rails'

**and** **c**.created\_at **>** DATE\_SUB(NOW(), INTERVAL 3 **MONTH**)

**union**

**select** **distinct**(u.login) **as** login

**from** pull\_requests pr, projects p, users u, users u1, pull\_request\_history prh

**where** u.id **=** prh.actor\_id

**and** prh.action **=** 'merged'

**and** u1.id **=** p.owner\_id

**and** prh.pull\_request\_id **=** pr.id

**and** pr.base\_repo\_id **=** p.id

**and** prh.created\_at **>** DATE\_SUB(NOW(), INTERVAL 3 **MONTH**)

**and** p.name **=** 'rails'

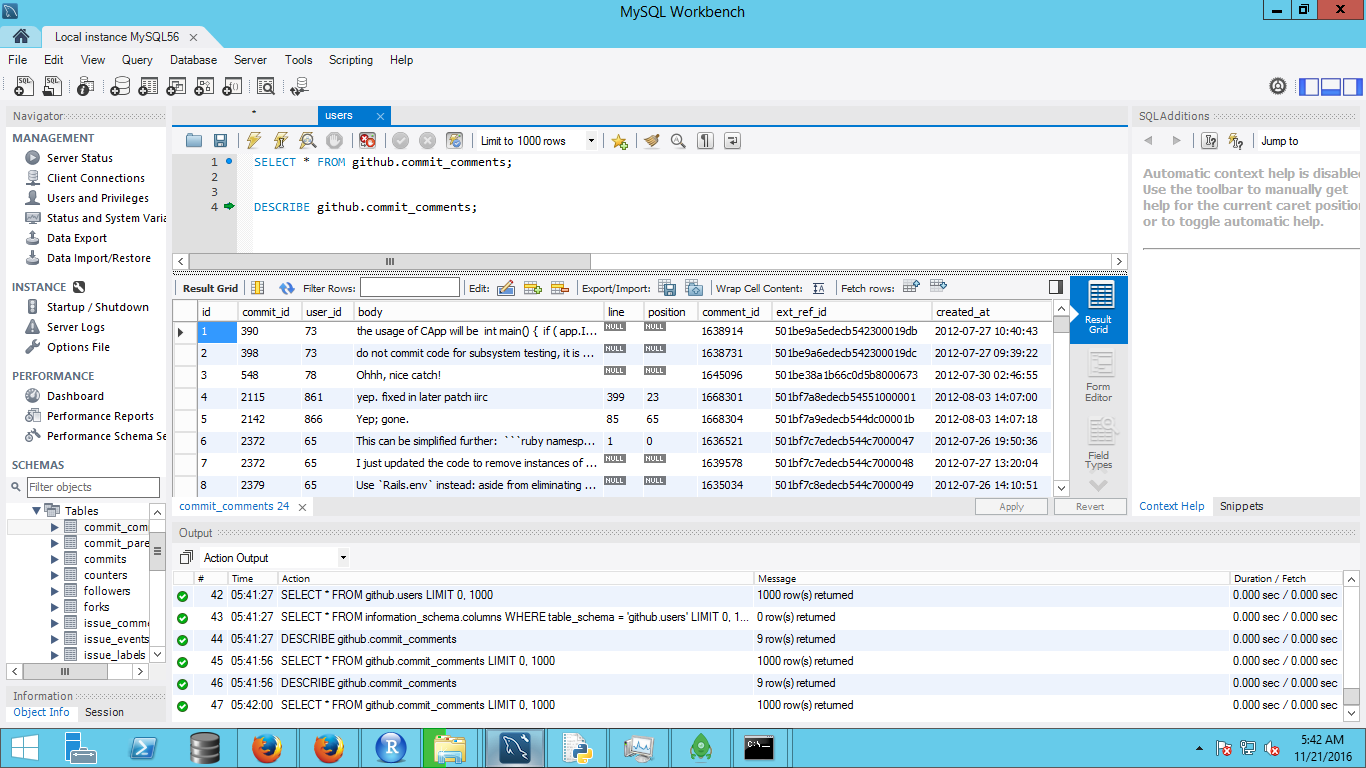
**and** u1.login **=** 'rails'

1. Users:

|  |
| --- |
| id |
| login |
| name |
| company |
| location |
| email |
| created\_at |
| ext\_ref\_id |
| type |
|  |

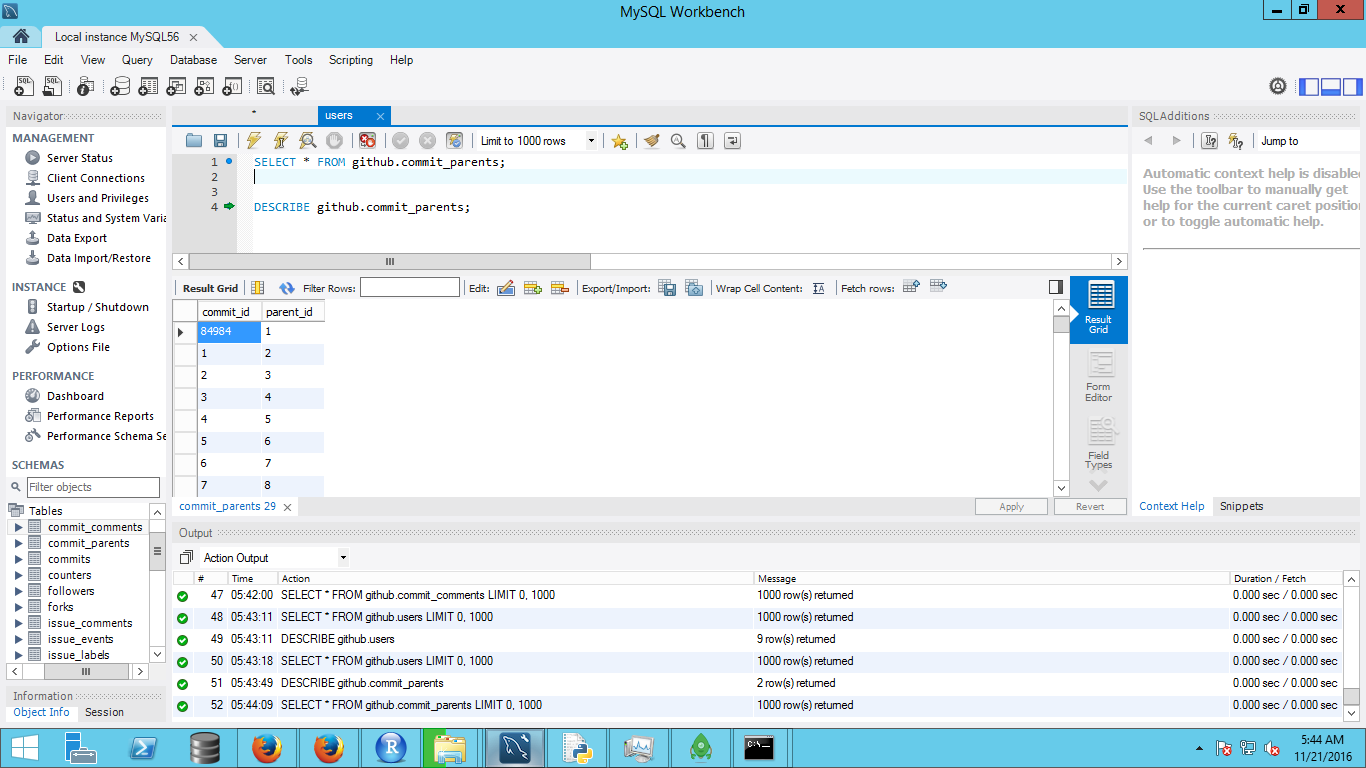
1. Commit\_comments

|  |
| --- |
| id |
| commit\_id |
| user\_id |
| body |
| line |
| position |
| comment\_id |
| ext\_ref\_id |
| created\_at |



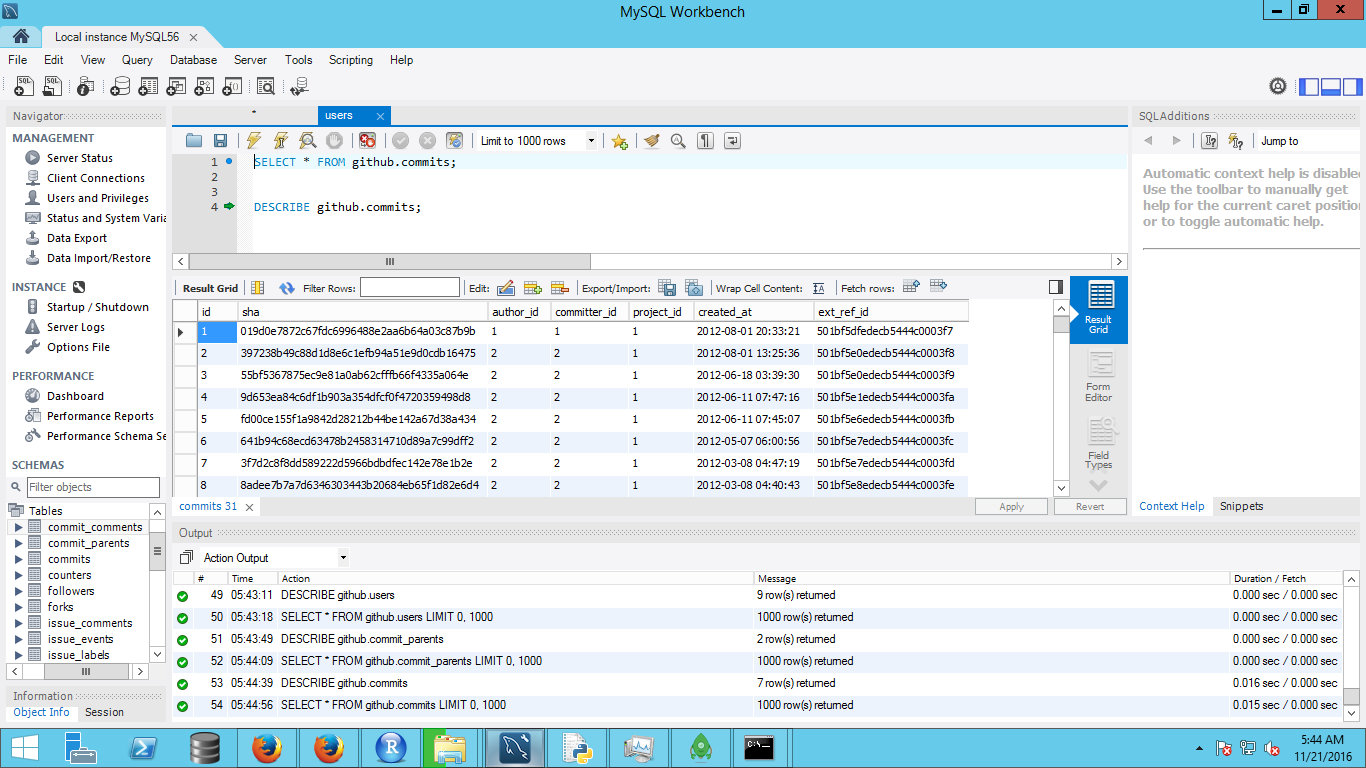
1. Commit\_parents

|  |
| --- |
| commit\_id |
| parent\_id |



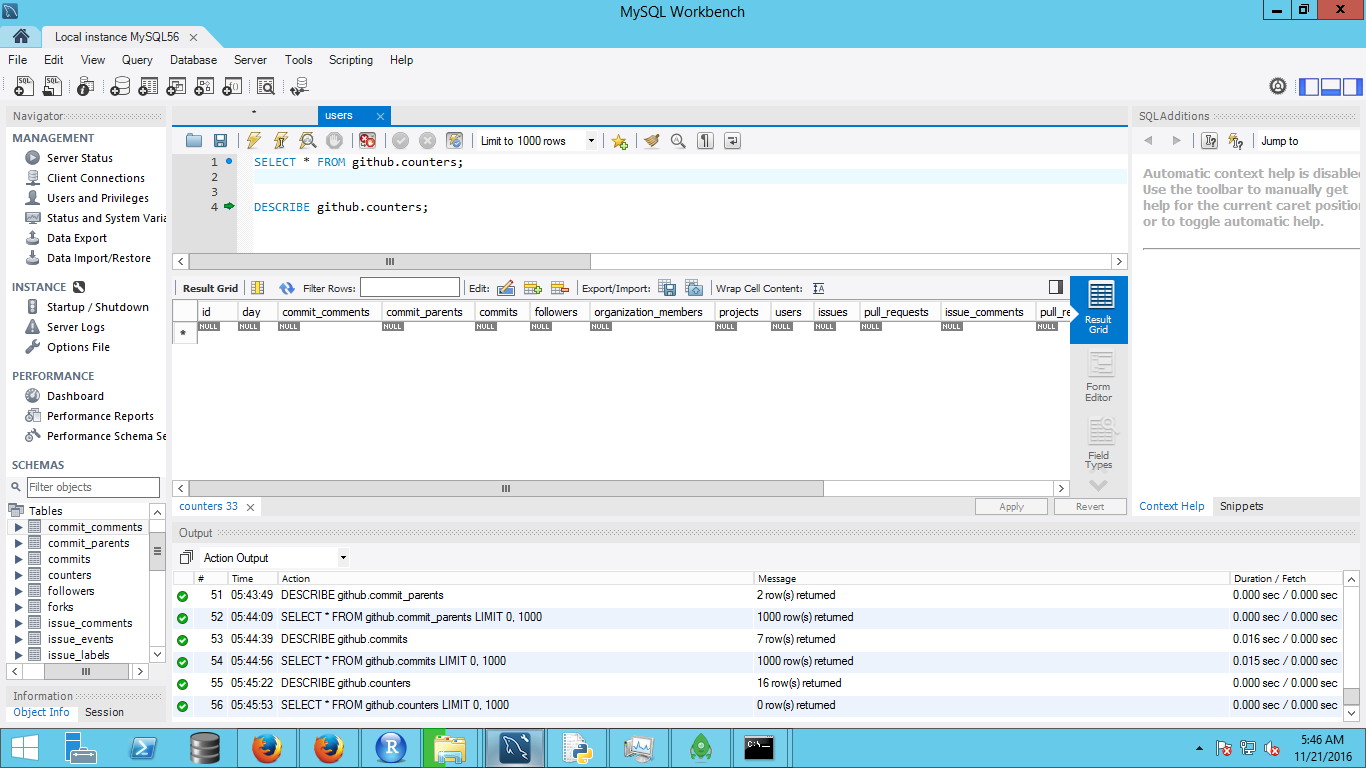
1. Commits

|  |
| --- |
| id |
| sha |
| author\_id |
| committer\_id |
| project\_id |
| created\_at |
| ext\_ref\_id |



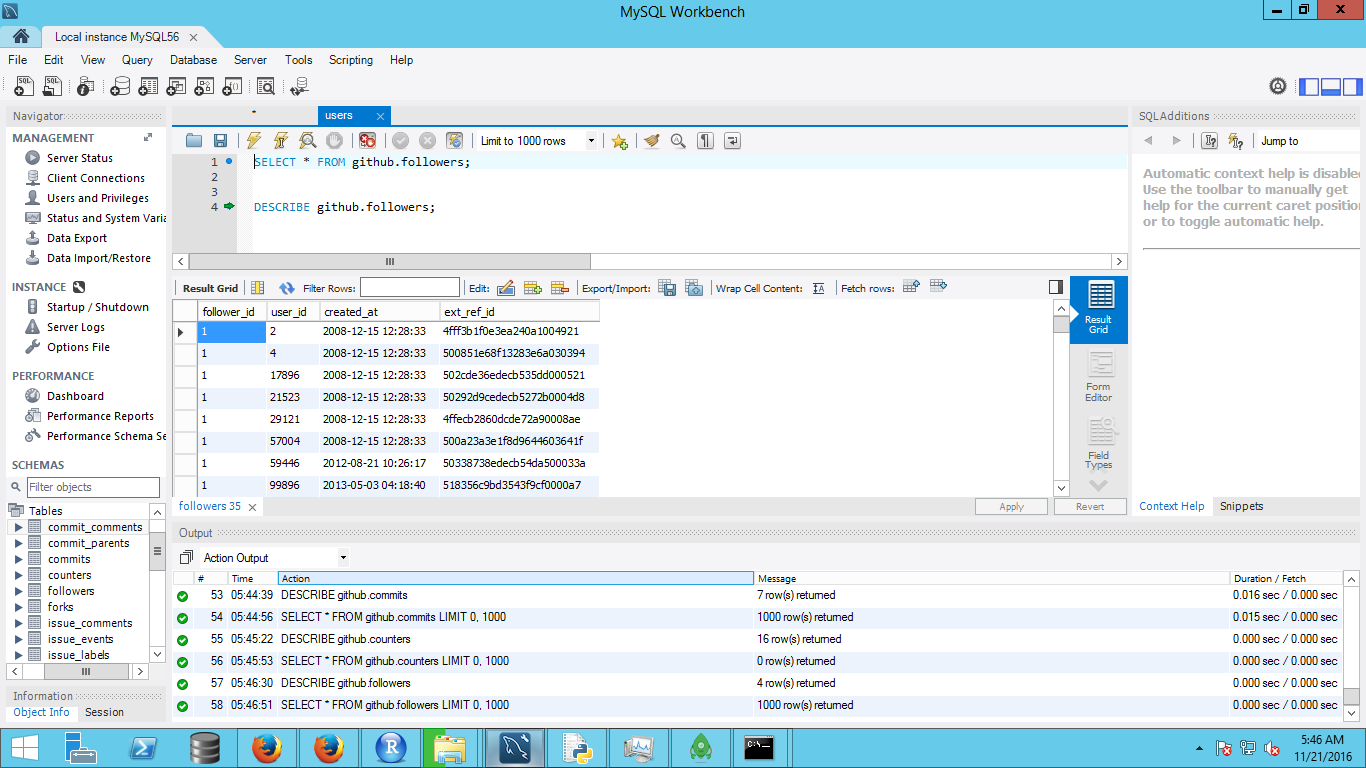
1. Counters

|  |
| --- |
| id |
| day |
| commit\_comments |
| commit\_parents |
| commits |
| followers |
| organization\_members |
| projects |
| users |
| issues |
| pull\_requests |
| issue\_comments |
| pull\_request\_comments |
| pull\_request\_history |
| watchers |
| forks |



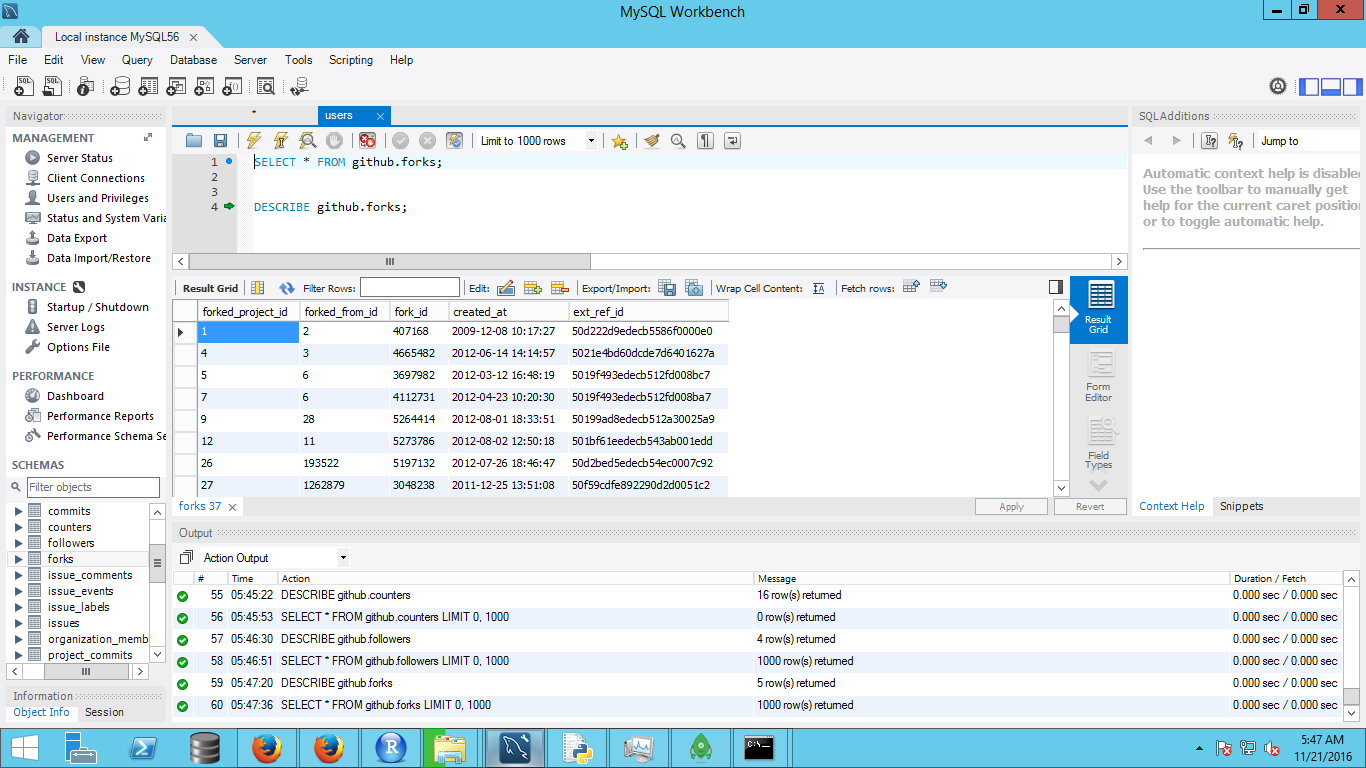
1. Followers

|  |
| --- |
| follower\_id |
| user\_id |
| created\_at |
| ext\_ref\_id |



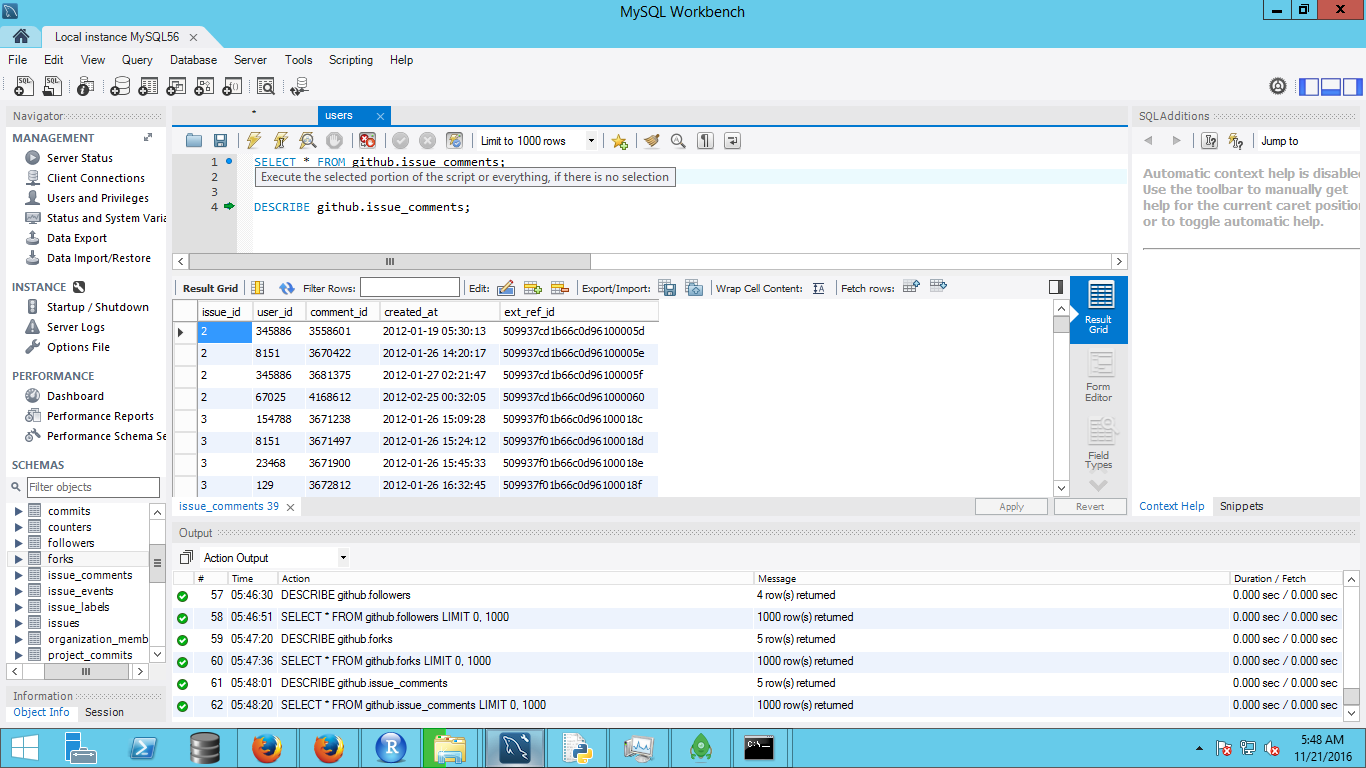
1. Forks

|  |
| --- |
| forked\_project\_id |
| forked\_from\_id |
| fork\_id |
| created\_at |
| ext\_ref\_id |



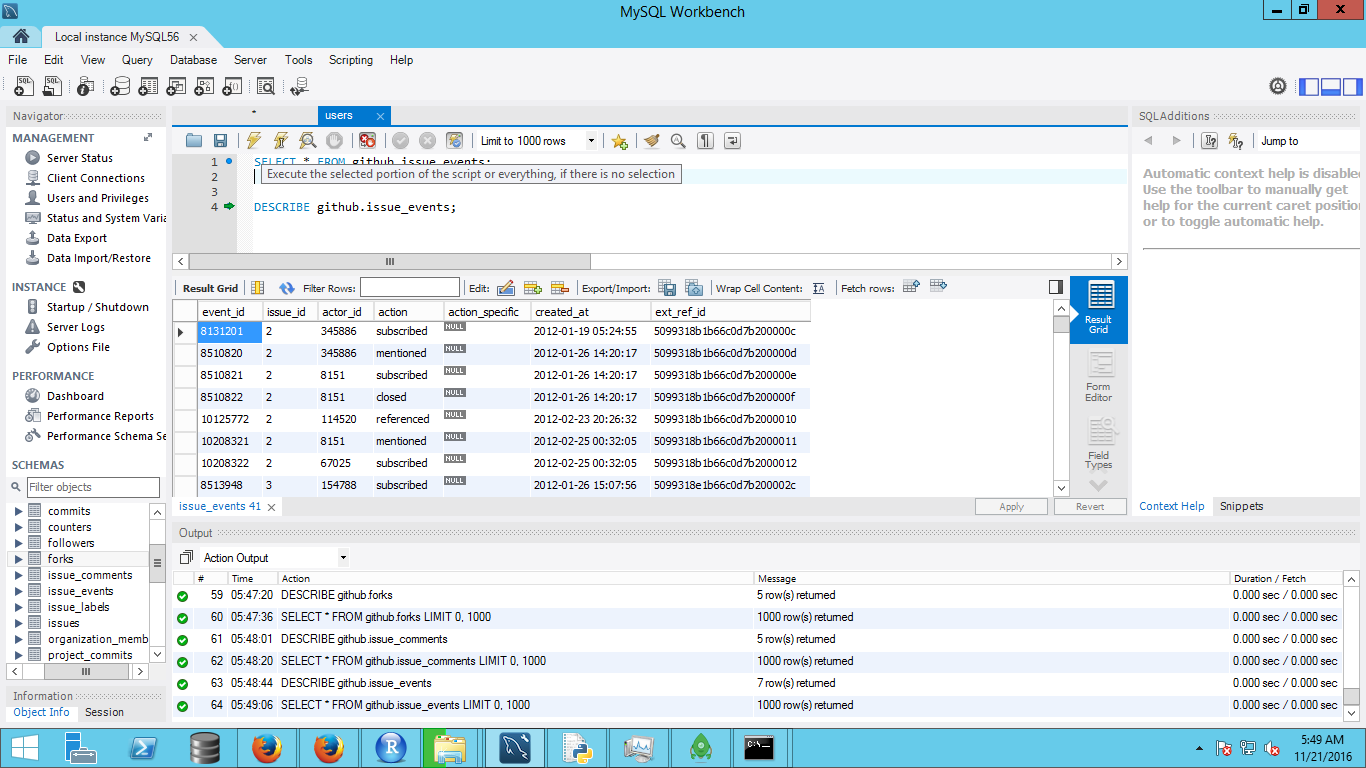
1. Issue\_comments

|  |
| --- |
| issue\_id |
| user\_id |
| comment\_id |
| created\_at |
| ext\_ref\_id |



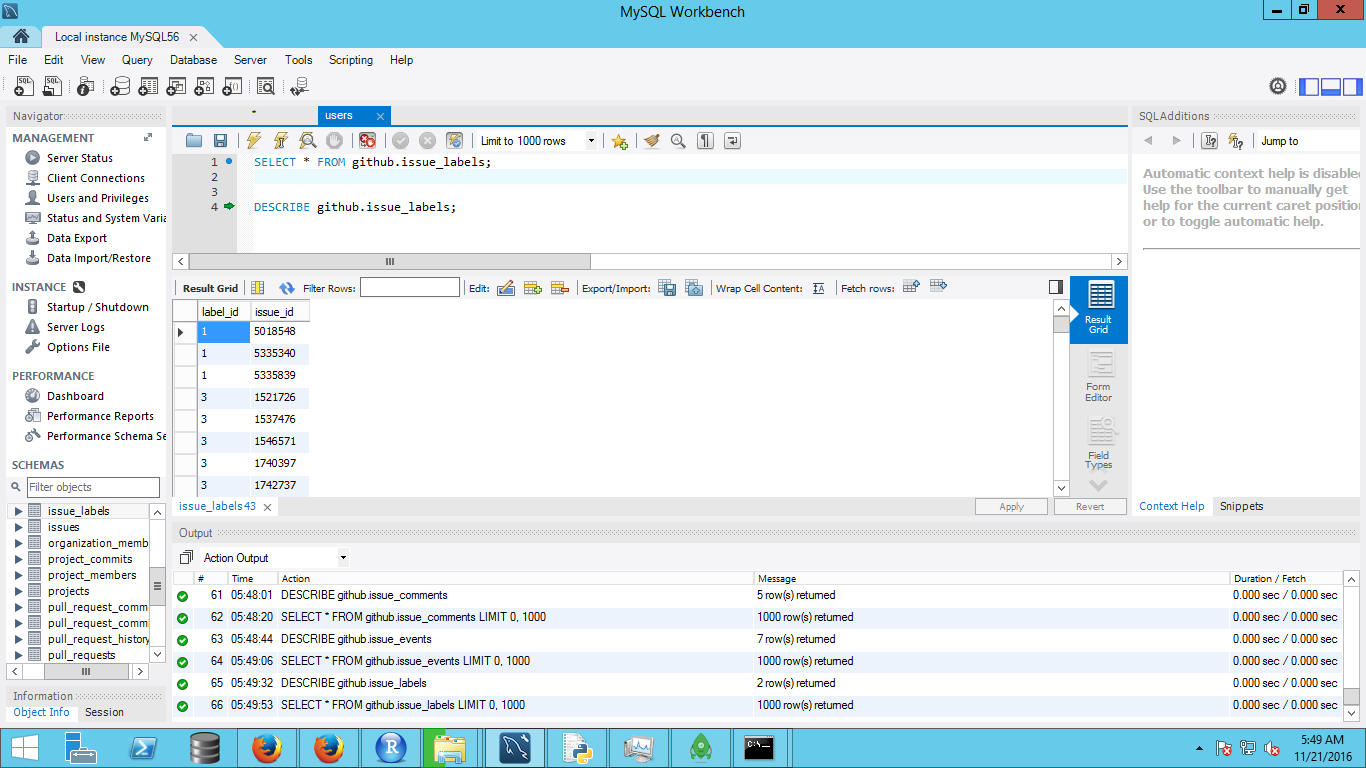
1. Issue\_events

|  |
| --- |
| event\_id |
| issue\_id |
| actor\_id |
| action |
| action\_specific |
| created\_at |
| ext\_ref\_id |



1. Issue\_labels

|  |
| --- |
| label\_id |
| issue\_id |

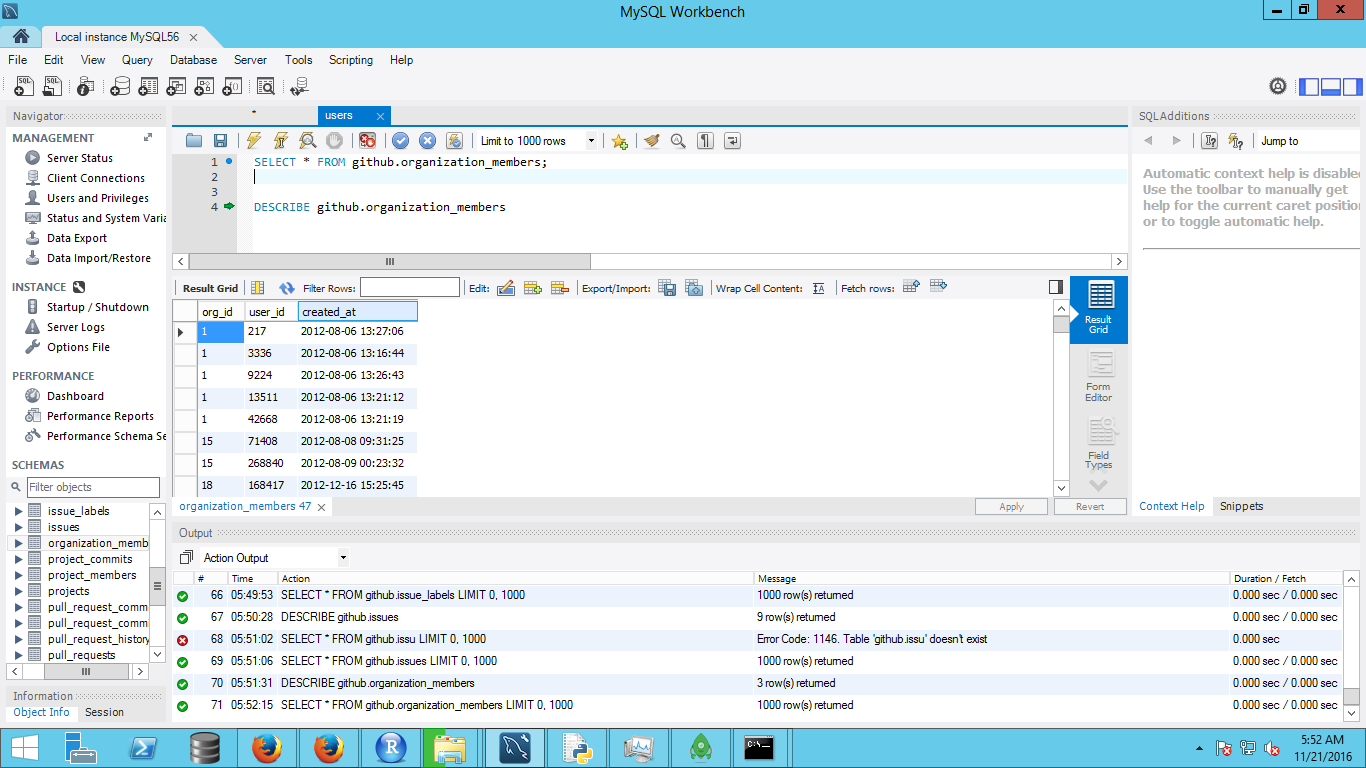


1. Issues

|  |
| --- |
| id |
| repo\_id |
| reporter\_id |
| assignee\_id |
| pull\_request |
| pull\_request\_id |
| created\_at |
| ext\_ref\_id |
| issue\_id |

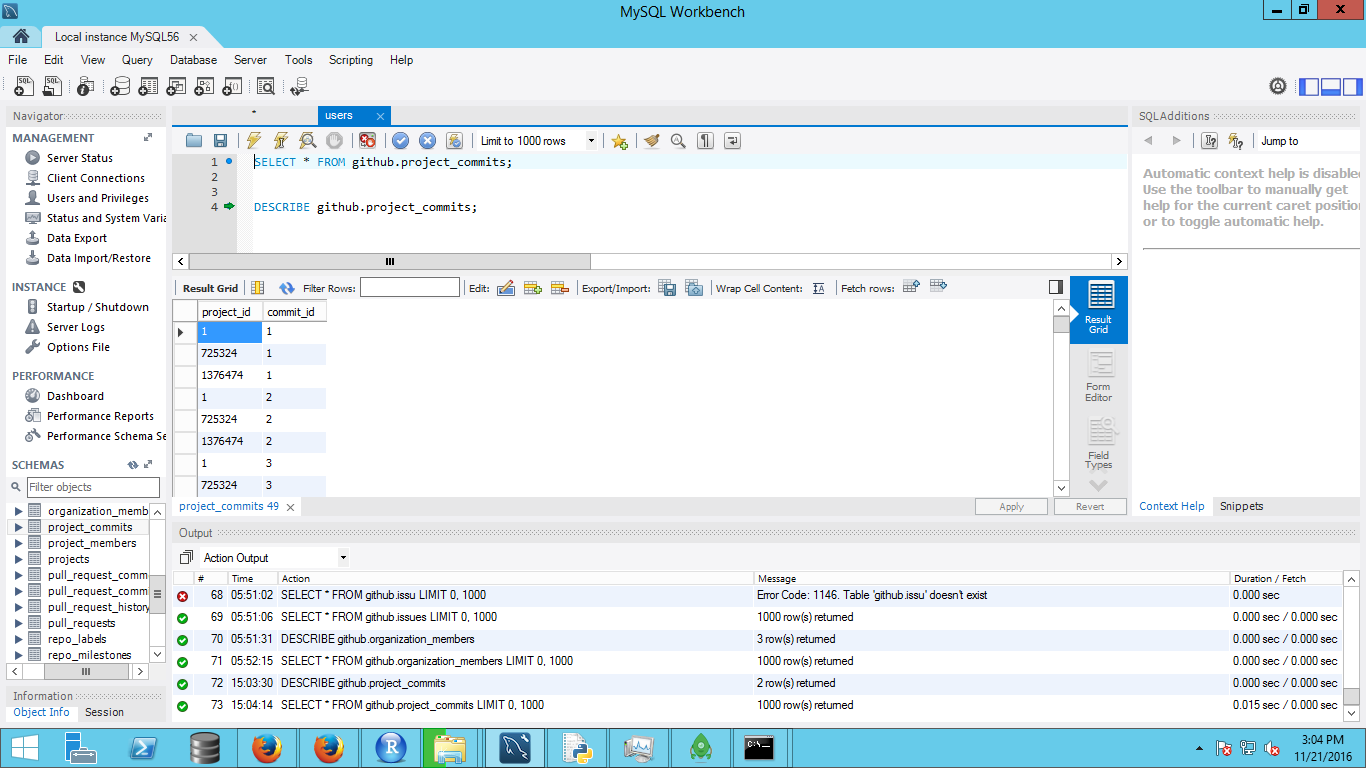
12. Organization members

|  |
| --- |
| org\_id |
| user\_id |
| created\_at |



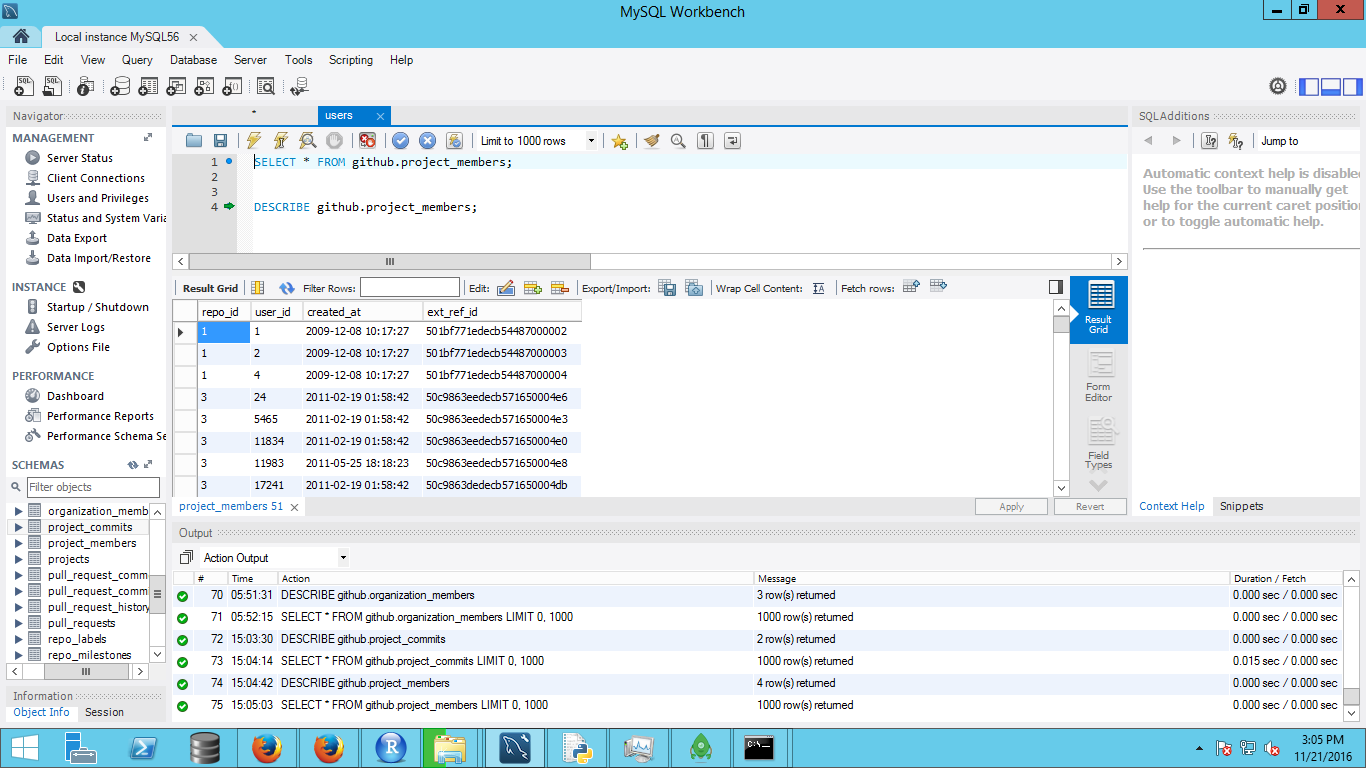
1. Project Commits

|  |
| --- |
| project\_id |
| commit\_id |



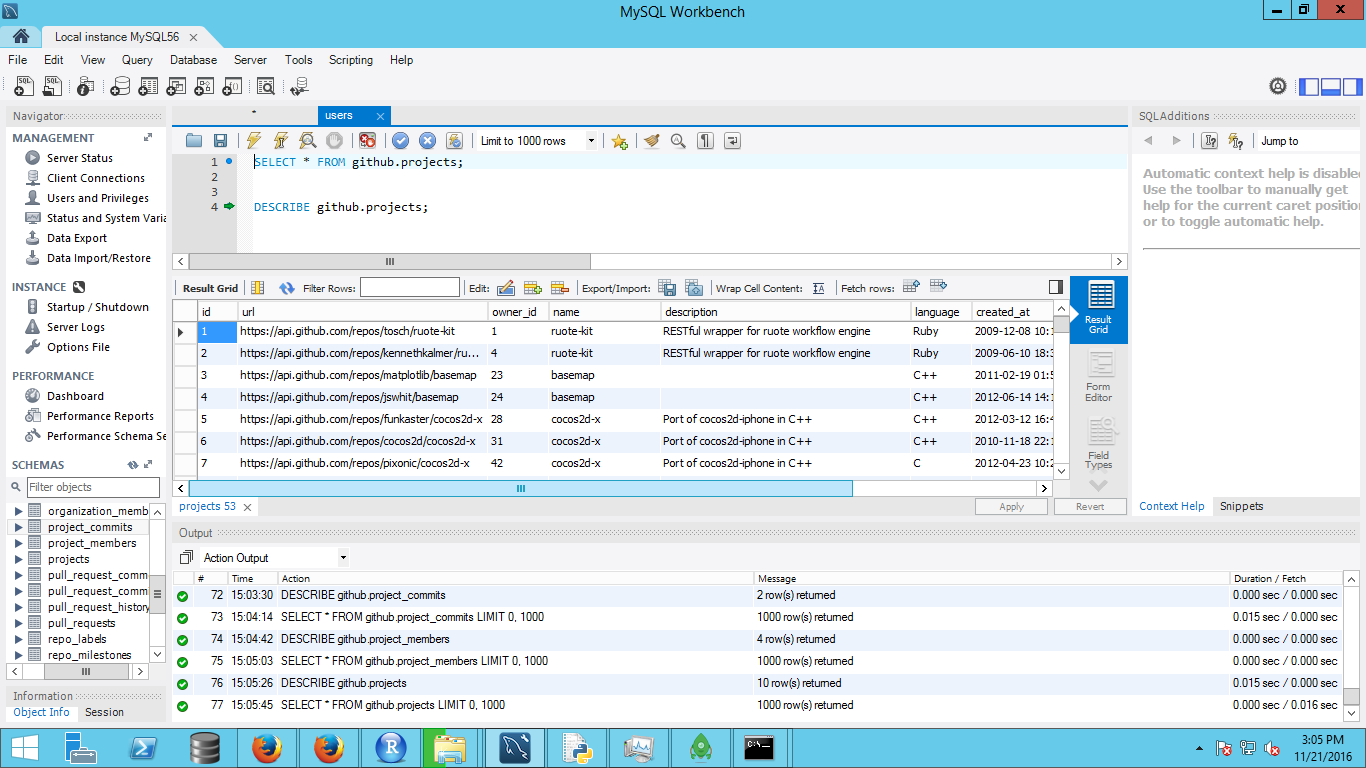
1. Project Members

|  |
| --- |
| repo\_id |
| user\_id |
| created\_at |
| ext\_ref\_id |



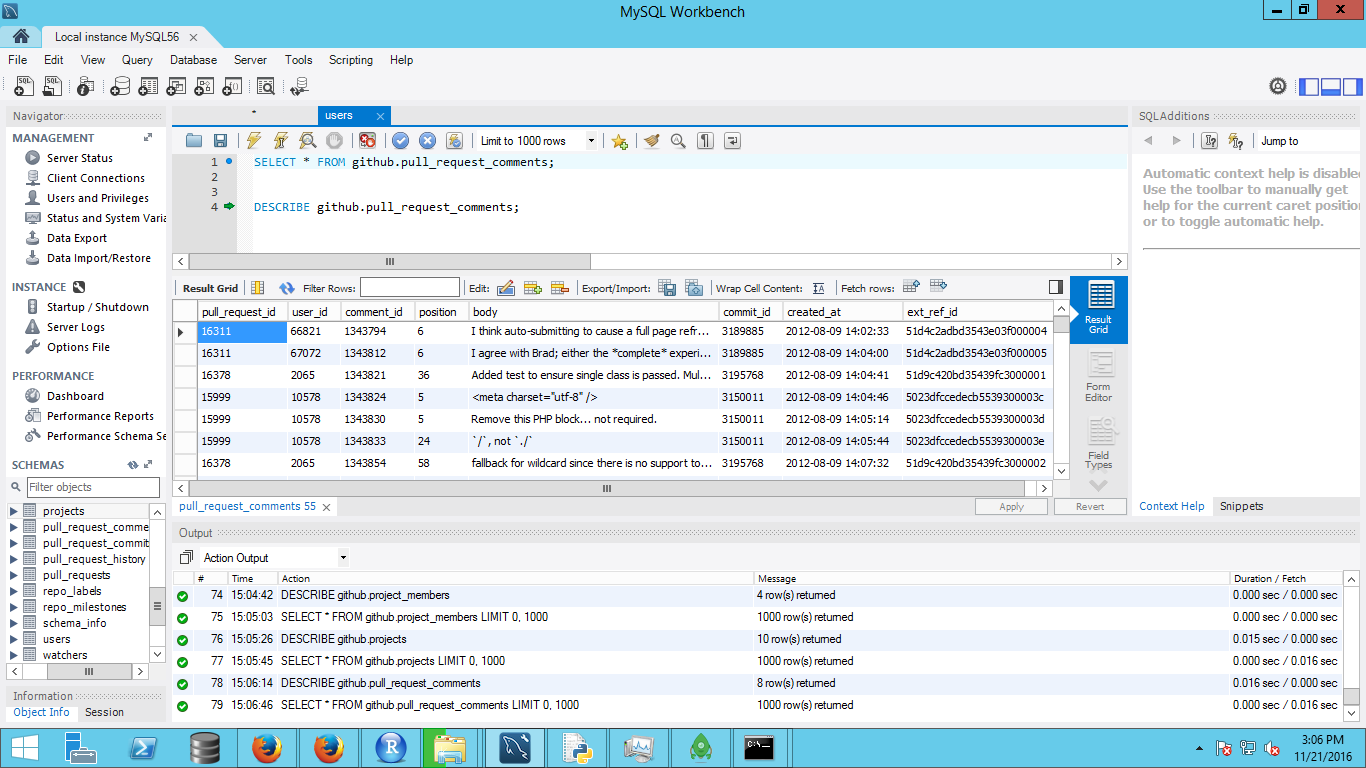
1. Projects

|  |
| --- |
| id |
| url |
| owner\_id |
| name |
| description |
| language |
| created\_at |
| ext\_ref\_id |
| forked\_from |
| deleted |



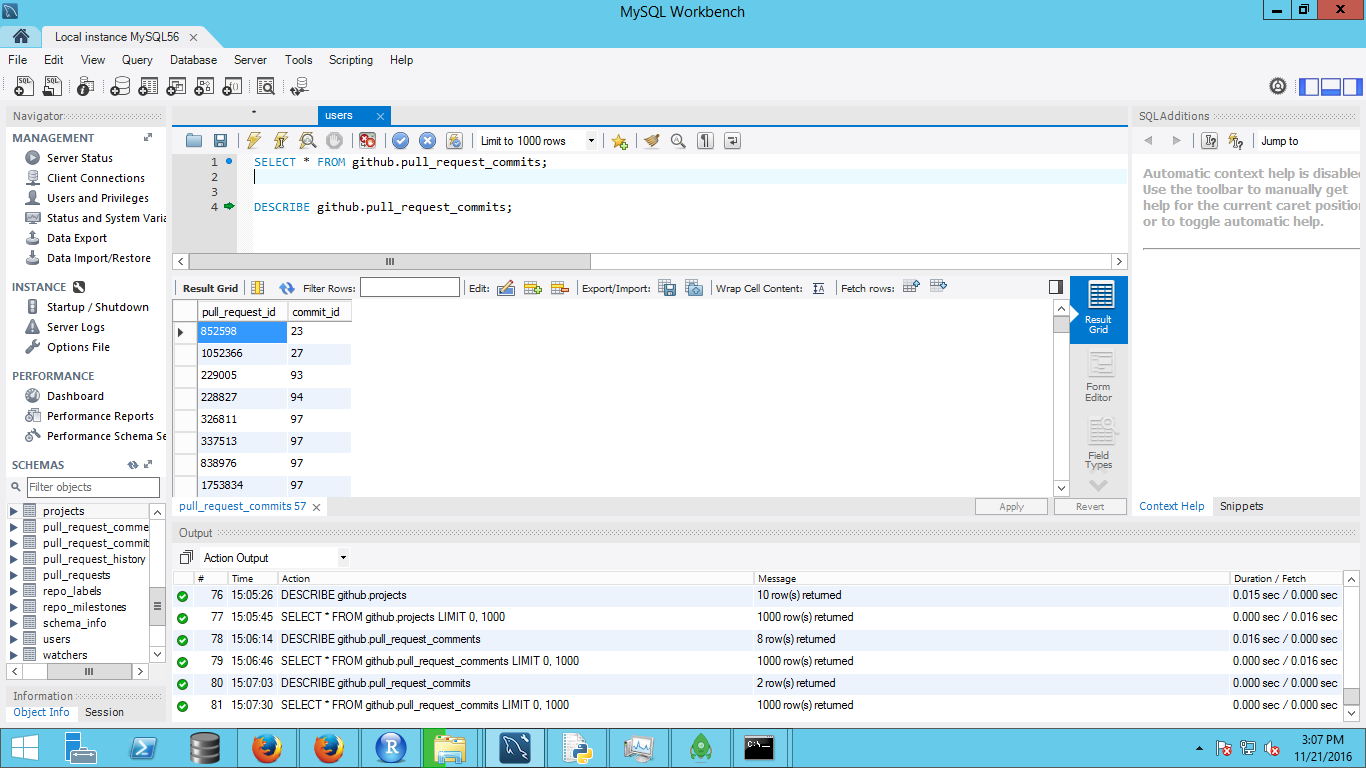
1. Pull Request Comments

|  |
| --- |
| pull\_request\_id |
| user\_id |
| comment\_id |
| position |
| body |
| commit\_id |
| created\_at |
| ext\_ref\_id |



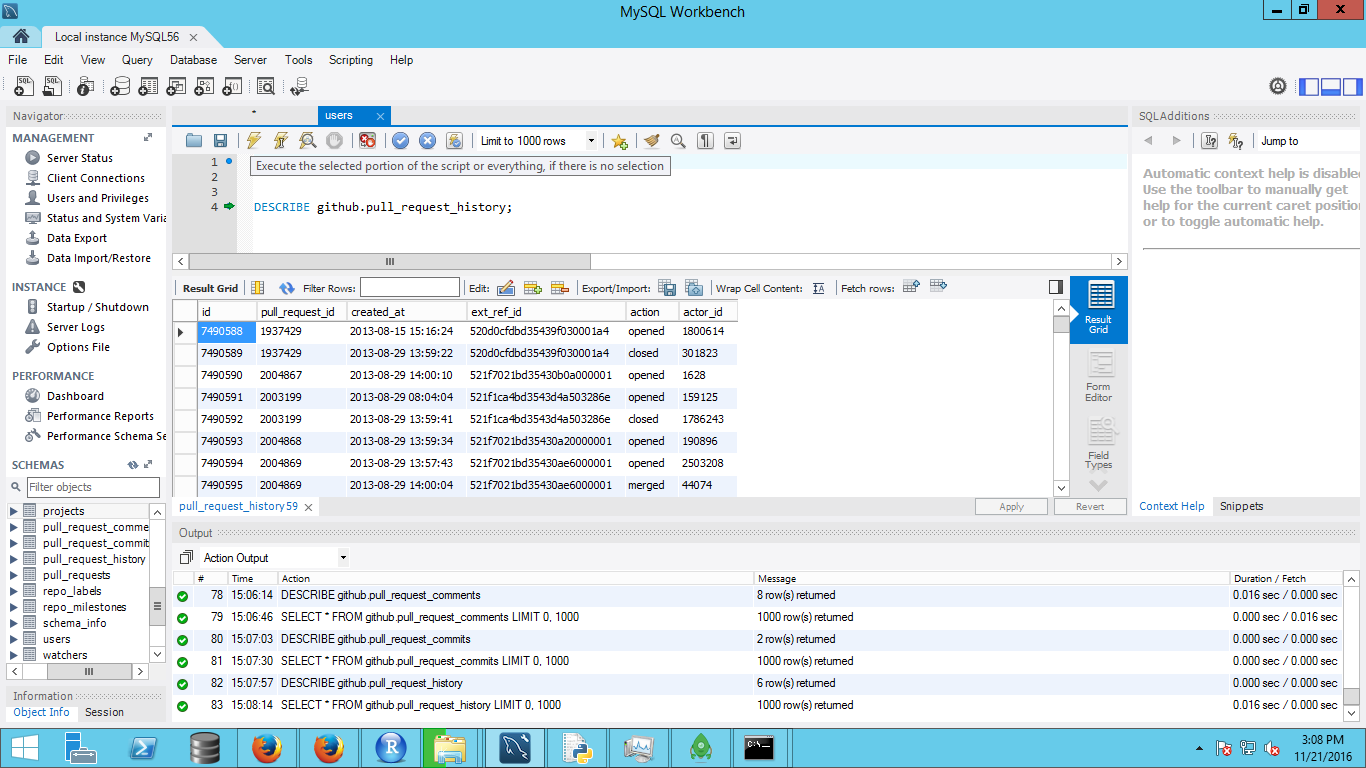
1. Pull Request Commits

|  |
| --- |
| pull\_request\_id |
| commit\_id |



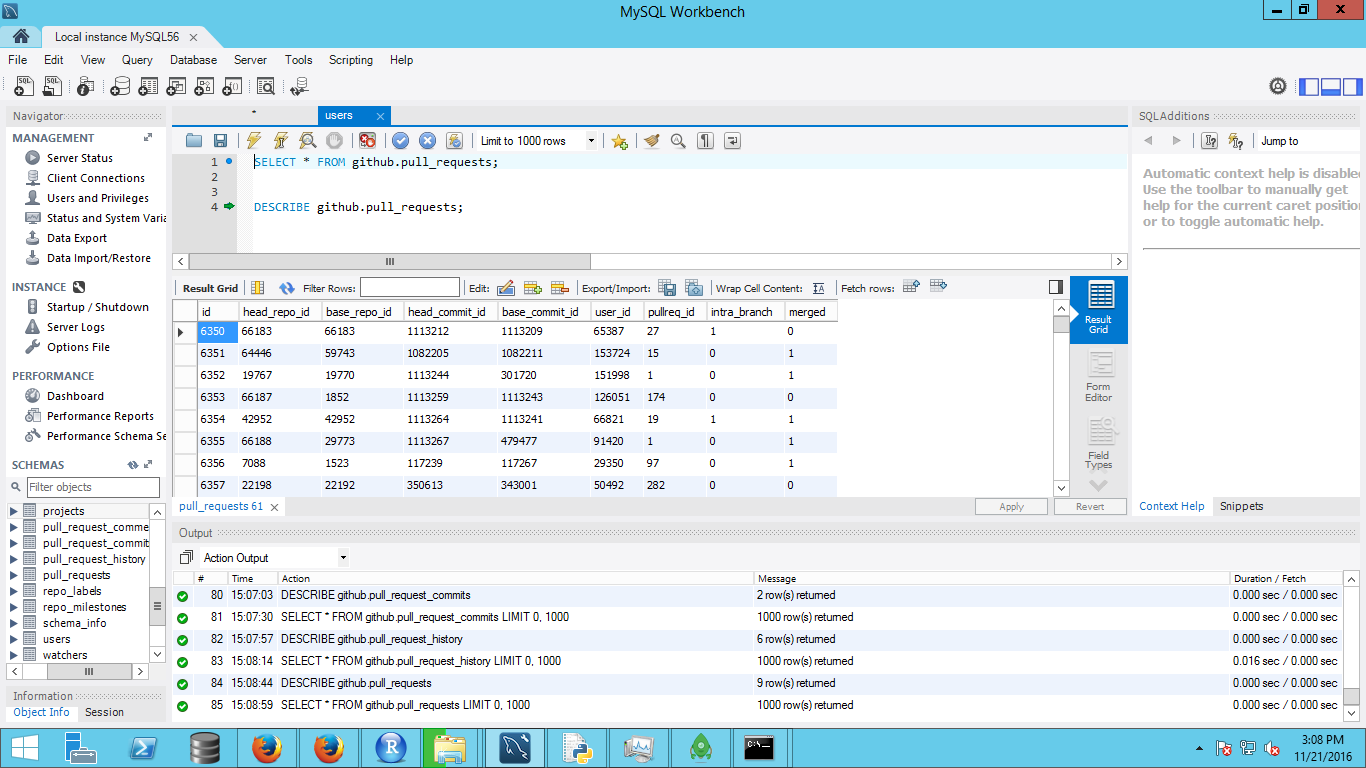
1. Pull Request History

|  |
| --- |
| id |
| pull\_request\_id |
| created\_at |
| ext\_ref\_id |
| action |
| actor\_id |



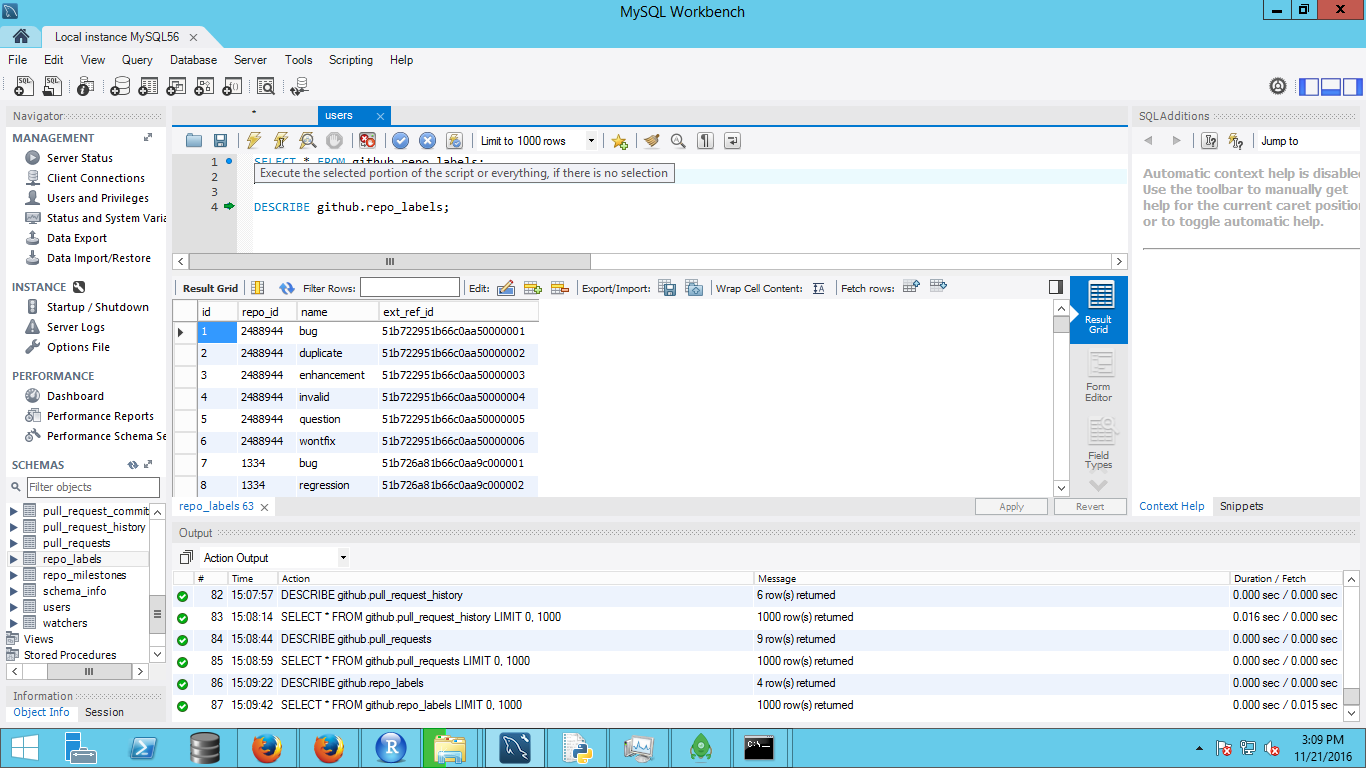
1. Pull Requests

|  |
| --- |
| id |
| head\_repo\_id |
| base\_repo\_id |
| head\_commit\_id |
| base\_commit\_id |
| user\_id |
| pullreq\_id |
| intra\_branch |
| merged |



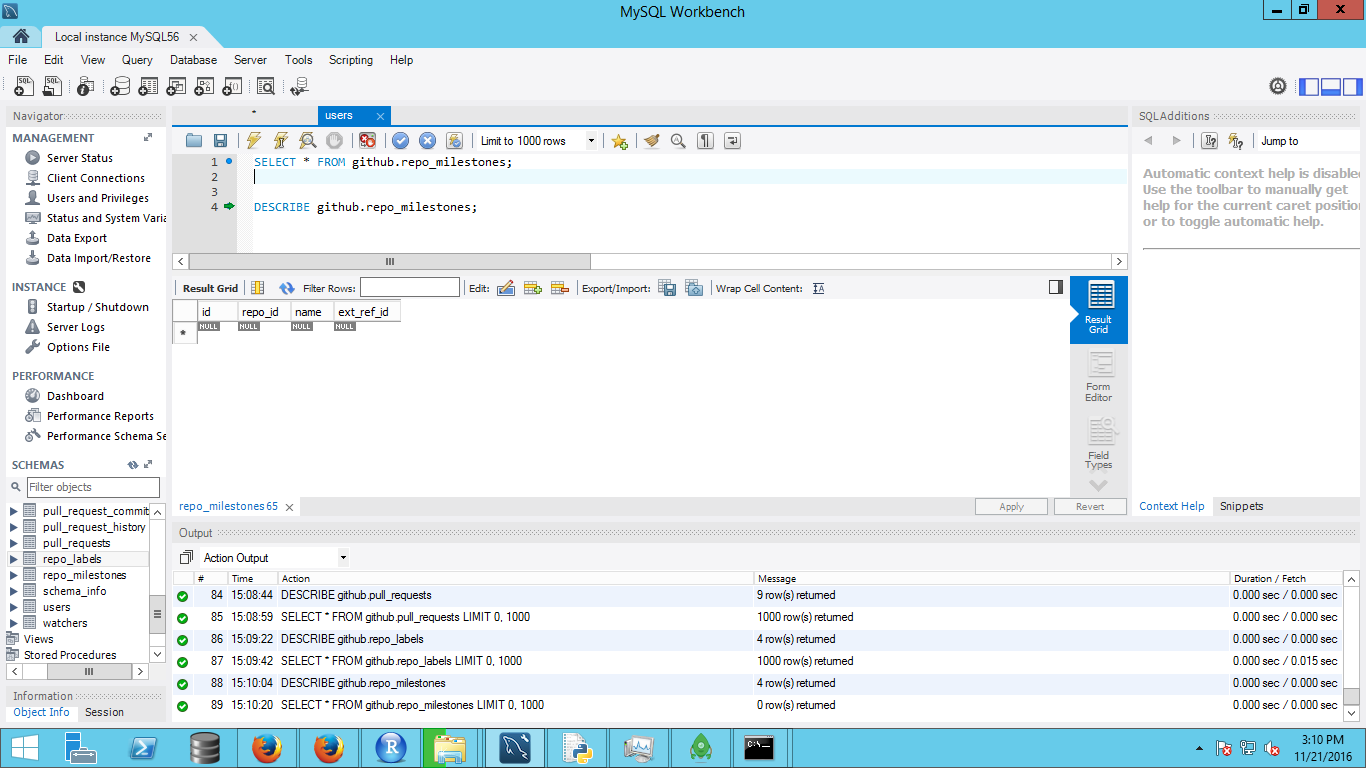
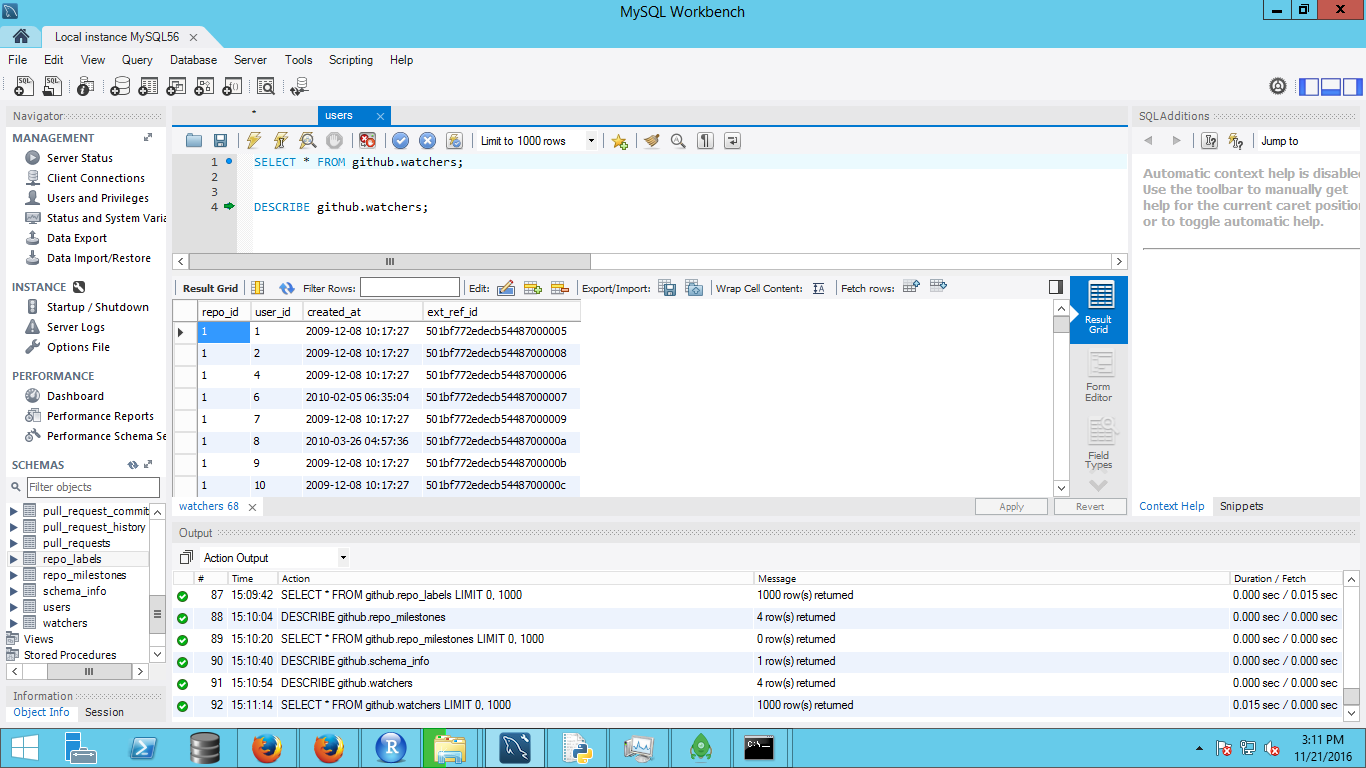
1. Repo Labels

|  |
| --- |
| id |
| repo\_id |
| name |
| ext\_ref\_id |



1. Repo Milestones

|  |
| --- |
| id |
| repo\_id |
| name |
| ext\_ref\_id |



1. Watchers

|  |
| --- |
| repo\_id |
| user\_id |
| created\_at |
| ext\_ref\_id |

**Exploratory analysis on the MYSQL data**

1. Select a single user from the table users:

SELECT \* FROM github.users where id=8137;

1. Group the user commit comments and order by the count in descending order

SELECT user\_id, count(user\_id) FROM github.commit\_comments group by user\_id order by 2 desc;

1. Commit parents

SELECT \* FROM github.commit\_parents;

1. Check where the author\_id and commiter\_id are not the same. Such as in merge requests

SELECT \* FROM github.commits where author\_id<>committer\_id;

1. Group the user commits and order by the count in descending order

SELECT author\_id, count(author\_id) FROM github.commits group by author\_id order by 2 desc;

1. Check counters. This table in null

SELECT \* FROM github.counters;

1. Group the user followers and order by the count in descending order

SELECT user\_id, count(follower\_id) FROM github.followers group by user\_id order by 2 desc;

1. Group the user following and order by the count in descending order

SELECT follower\_id, count(user\_id) FROM github.followers group by follower\_id order by 2 desc;

1. Check the forks

SELECT \* FROM github.forks;

1. Group the user issue comments and order by the count in descending order

SELECT user\_id, count(\*) FROM github.issue\_comments group by user\_id order by 2 desc;

1. Group the user event actions where the user subscribed to any issue and order by the count in descending order

select actor\_id as user\_id, count(\*) from issue\_events where action = "subscribed" group by user\_id order by 2 desc;

1. Group the user event actions where the user was mentioned in any issue and order by the count in descending order

select actor\_id as user\_id, count(\*) from issue\_events where action = "mentioned" group by user\_id order by 2 desc;

1. Group the user event actions where the user unsubscribed to any issue and order by the count in descending order

select actor\_id as user\_id, count(\*) from issue\_events where action = "unsubscribed" group by user\_id order by 2 desc;

1. Group the user issues and order by the count in descending order

select reporter\_id, count(\*) from issues group by reporter\_id order by 2 desc;

1. Group the user organizations and order by the count in descending order

select user\_id, count(\*) from organization\_members group by user\_id order by 2 desc;

1. Group the user issue comments and order by the count in descending order

select user\_id, count(\*) from project\_members group by user\_id order by 2 desc;

1. Explore the projects

select \* from projects;

1. Group the user pull request comments and order by the count in descending order

select user\_id, count(\*) from pull\_request\_comments group by user\_id order by 2 desc;

1. Explore pull requet commits

select \* from pull\_request\_commits;

1. Explore pull request history

select \* from pull\_request\_history;

1. Group the user pull requests and order by the count in descending order

select user\_id, count(\*) from pull\_requests group by user\_id order by 2 desc;

1. Group the watchers and order by the count in descending order

select user\_id, count(\*) from watchers group by user\_id order by 2 desc;

**Create a table for the 1000 users**

1. Create a subset of 1000 users for sub-qeuries

CREATE TABLE `1000users` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`login` varchar(255) NOT NULL,

`name` varchar(255) DEFAULT NULL,

`company` varchar(255) DEFAULT NULL,

`location` varchar(255) DEFAULT NULL,

`email` varchar(255) DEFAULT NULL,

`created\_at` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

`ext\_ref\_id` varchar(24) NOT NULL DEFAULT '0',

`type` varchar(255) NOT NULL DEFAULT 'USR',

PRIMARY KEY (`id`),

UNIQUE KEY `login` (`login`),

KEY `users\_email` (`email`)

) ENGINE=InnoDB AUTO\_INCREMENT=2672718 DEFAULT CHARSET=utf8;

1. Insert the data for 1000 unique users

insert into 1000users (select \* from users limit 1000);

1. Join multiple tables and get the data for 1000 users

select u.\*, c.\*, c2.\*,c3.\*, c4.\*, c5.\*, c6.\*, c7.\*, c8.\*, c9.\*, c10.\*, c11.\*, c12.\*, c13.\*, c14.\* from

(select \* from 1000users ) u

left join (SELECT user\_id, count(user\_id) as 'commit\_comments' FROM github.commit\_comments where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c

on u.id=c.user\_id

left join (SELECT author\_id as user\_id, count(author\_id) as 'commits' FROM github.commits where author\_id in (select id from 1000users) group by author\_id order by 2 desc) c2

on u.id=c2.user\_id

left join (SELECT user\_id, count(follower\_id) as 'followers' FROM github.followers where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c3

on u.id=c3.user\_id

left join (SELECT follower\_id as user\_id, count(user\_id) as 'following' FROM github.followers where follower\_id in (select id from 1000users) group by follower\_id order by 2 desc) c4

on u.id = c4.user\_id

left join (SELECT user\_id, count(\*) as 'issue\_coments' FROM github.issue\_comments where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c5

on u.id = c5.user\_id

left join (select actor\_id as user\_id, count(\*) as 'subscribed' from issue\_events where action = "subscribed" and actor\_id in (select id from 1000users) group by user\_id order by 2 desc) c6

on u.id = c6.user\_id

left join (select actor\_id as user\_id, count(\*) as 'mentioned' from issue\_events where action = "mentioned" and actor\_id in (select id from 1000users) group by user\_id order by 2 desc) c7

on u.id = c7.user\_id

left join (select actor\_id as user\_id, count(\*) as 'ubsubscribed' from issue\_events where action = "unsubscribed" and actor\_id in (select id from 1000users) group by user\_id order by 2 desc) c8

on u.id = c8.user\_id

left join (select reporter\_id as user\_id, count(\*) as 'issues' from issues where reporter\_id in (select id from 1000users) group by reporter\_id order by 2 desc) c9

on u.id = c9.user\_id

left join(select user\_id, count(\*) as 'org\_members' from organization\_members where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c10

on u.id = c10.user\_id

left join(select user\_id, count(\*) as 'project\_members' from project\_members where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c11

on u.id = c11.user\_id

left join (select user\_id, count(\*) as 'pull\_request\_comments' from pull\_request\_comments where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c12

on u.id = c12.user\_id

left join (select user\_id, count(\*) as 'pull\_requests' from pull\_requests where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c13

on u.id = c13.user\_id

left join (select user\_id, count(\*) as 'watching' from watchers where user\_id in (select id from 1000users) group by user\_id order by 2 desc) c14

on u.id = c14.user\_id;

1. Create a table for these users to store their data

CREATE TABLE `usersData` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`login` varchar(255) NOT NULL,

`name` varchar(255) DEFAULT NULL,

`company` varchar(255) DEFAULT NULL,

`location` varchar(255) DEFAULT NULL,

`email` varchar(255) DEFAULT NULL,

`created\_at` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

`ext\_ref\_id` varchar(24) NOT NULL DEFAULT '0',

`type` varchar(255) NOT NULL DEFAULT 'USR',

`commit\_comments` int(11) ,

`commits` int(11) ,

`followers` int(11) ,

`following` int(11) ,

`issue\_coments` int(11) ,

`subscribed` int(11) ,

`mentioned` int(11) ,

`ubsubscribed` int(11) ,

`issues` int(11) ,

`org\_members` int(11) ,

`project\_members` int(11) ,

`pull\_request\_comments` int(11) ,

`pull\_requests` int(11) ,

`watching` int(11) ,

PRIMARY KEY (`id`),

UNIQUE KEY `login` (`login`),

KEY `users\_email` (`email`)

) ENGINE=InnoDB AUTO\_INCREMENT=2672718 DEFAULT CHARSET=utf8;

CREATE TABLE `usersData` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`login` varchar(255) NOT NULL,

`name` varchar(255) DEFAULT NULL,

`company` varchar(255) DEFAULT NULL,

`location` varchar(255) DEFAULT NULL,

`email` varchar(255) DEFAULT NULL,

`created\_at` timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

`ext\_ref\_id` varchar(24) NOT NULL DEFAULT '0',

`type` varchar(255) NOT NULL DEFAULT 'USR',

`commit\_comments` int(11) ,

`commits` int(11) ,

`followers` int(11) ,

`following` int(11) ,

`issue\_coments` int(11) ,

`subscribed` int(11) ,

`mentioned` int(11) ,

`ubsubscribed` int(11) ,

`issues` int(11) ,

`org\_members` int(11) ,

`project\_members` int(11) ,

`pull\_request\_comments` int(11) ,

`pull\_requests` int(11) ,

`watching` int(11) ,

PRIMARY KEY (`id`),

UNIQUE KEY `login` (`login`),

KEY `users\_email` (`email`)

) ENGINE=InnoDB AUTO\_INCREMENT=2672718 DEFAULT CHARSET=utf8;