JGit blame for fun and profit(?)

Description	Analysis of git commits on OOS projects
Period	Fall 2017
Languages & Libs	Clojure
Tags	Git, Elasticsearch

Software projects are typically "tracked" on a version control system (VCS). Each "version" of the code is called a "commit", which does not only store file contents, but also plenty of metadata. This creates a very rich set of data, and in the age of open source there are thousands of projects to study. A few examples are Git of Theseus and Gitential, but by focusing on "`git blame`" (see who has committed each line on each file) I hope to bring something new to the table. In short I have analyzed how source code gets replaced by newer code, tracking the topics of who, when and why, and how old the code was.

Git's commit log is a directed acyclic graph (DAG), which can be visualized by gitk (an example is shown at Figure 1). A commit can have arbitrary many parents and children, but the most typical cases are just one or two. Commits from which a new branch has been created have more than one child, and "merge" commits have more than one parent. As in real life parents are "older" than children, and the list of a commit's parents is immutable but new children might (unexpectedly or not) appear.

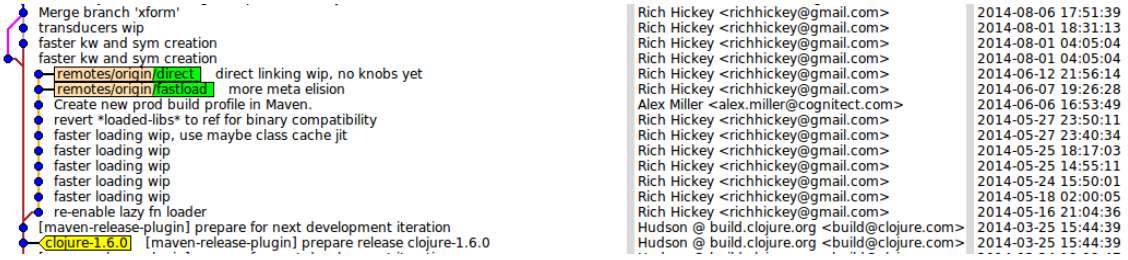

Figure 1: Clojure project's commits visualized by gitk.

Git doesn't actually store commits as "deltas" from the previous version, but instead refers to complete folders and files (for details have a look at Git internals). This structure has some interesting properties, for example when calculating the difference between two commits once can skip entire sub-directories by just determining that tree-objects' hashes are identical. When two files differ, the end user typically wants to ignore the identical parts and just focus on what was changed (a "patch"). Calculating this representation is actually a surprisingly complex task, some algorithms are explained at StackOverflow. An example commit visualization on `gitk` is shown at Figure 2.

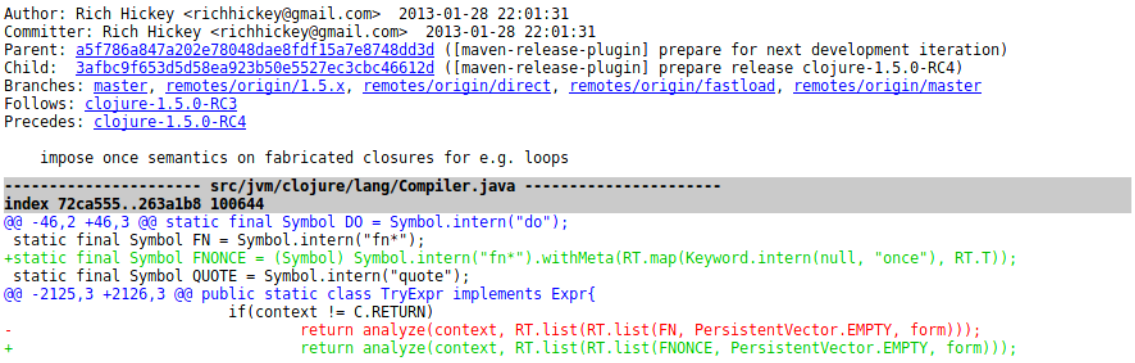

Figure 2: An example commit from Clojure's repository (commit 9b80a552fdabe...).

In this project I used JGit to extract information from a repository (aka. "repo"), but as I was using Clojure I was happy to find a JGit wrapper clj-jgit for it. At the time of writing my source code is only about 200 lines of code long, but it had to deal with some complexities such as how to cache intermediate results to speed up development and re-runs, indentifying nuances such as JGit throwing a null pointer exception when blaming a symlink, how to represent files "blames" and how to apply patches to them, and how to store all the results to Elasticsearch for future analysis.

The first step on a repository's analysis is to load it from disk to memory via the `jgit/load-repo`, which sets the context for all future API calls such as `jgit/git-blame`. The second step is to find all commits since a given date (as we might want to focus on last X years of the project), and find all "root" commits. These are commits which have no parents, or all of their parents were created before the chosen cutoff date. From these root commits we can find all commits which follow, and partition them into a linear sequences (each commit has exactly one parent) what I decided to call "hash-chains". I decided to ignore merge commits on this analysis, although they are critical points in branches lives. Merges do not create new code per se, unless there is a merge conflict.

At this stage we are closer to running the actual analysis. Calculating `git blame` on a large repo turns out to be a quite expensive operation, because it needs to determine for all lines in all files that from which commit it originates. Luckily it is sufficient to calculate it only for root commits (instead of all of them) and only for files which have changes within the commits we are interested of. In addition it is trivial to parallelize. Calculating patches between two commits is orders of magnitude faster, so this analysis can be run reasonably fast by using "root blames" as a starting point and using incremental patches to determine how the "blame state" looks like at subsequent commits. Putting all this together (and some post-processing) results in a sequence of hash-maps with a very rich set of information, an example is shown at Figure 3. It has the chunk's old and new commits' metadata (date, folder, file name + type, author's name and email and commit's hash + message) and derived information including the age of the code and if the new author is the same as the old one. The "id" of the data item is generated in a deterministic manner from commit hashes and the file path so that re-indexing all data to Elasticsearch will not leave old data behind, but will instead overwrite it.


```
{:age-days 1969.1397, :age-days-log2 10.9433, :old-time 1189269616000, :new-time 1359403291000,
 :full-path "src/jvm/clojure/lang/Compiler.java", :file-name "Compiler.java", :file-type ".java",
 :old-hash "d3ba5541568cd748daae80a9866fa109ded2d520", :new-hash "9b80a552fdabeabdd93951a625b55ae49c2f8d83", :id "e8ef34abe2ee93df8e950989d0df2db437dc8090",
 :new-auth-email "richhickey@gmail.com", :new-auth-name "Rich Hickey", :new-comm-email "richhickey@gmail.com", :new-comm-name "Rich Hickey",
 :old-auth-email "richhickey@gmail.com", :old-auth-name "Rich Hickey", :old-comm-email "richhickey@gmail.com", :old-comm-name "Rich Hickey",
 :new-comm-is-auth 1, :old-auth-is-new-auth 1, :old-comm-is-auth 1, :old-comm-is-new-comm 1,
 :old-msg "added dispatch map for parsers\n", :old-msg-tokens ("added" "dispatch" "for" "map" "parsers"),
 :new-msg "impose once semantics on fabricated closures for e.g. loops\n", :new-msg-tokens ("closures" "fabricated" "for" "impose" "loops" "on" "once" "semantics")}
```


Figure 3: An example hash-map, showing data on rows on a file src/jvm/clojure/lang/Compiler.java from the commit d3ab... being replaced by newer code from the commit 9b80a..., ending its 1969-days long contribution to Clojure.

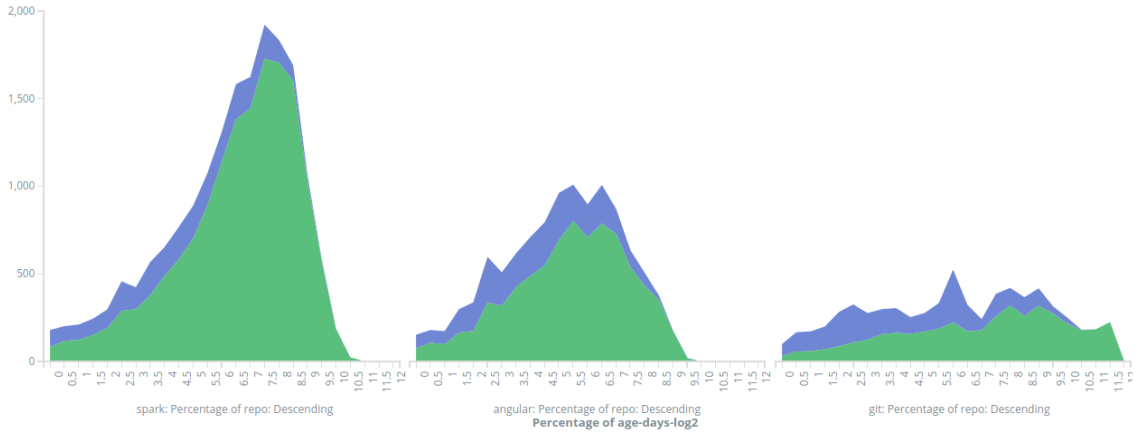

Figure 4: The age distribution of replaced code, with a color indication on "re-authorship".

Naturally one can simply keep all data in-memory and run any analysis on it, but Elasticsearch + Kibana has been proven to be a very scalable and relatively easy-to-use solution for dashboards and ad-hoc analysis and search. Thus I decided to use it on this project as well. I can think of dozens of analyses to test on this, but at this time I'll just describe a very simple one. We all might commit buggy code and push it every now and then, and when this happens we rush to fix the mistake. Also typically when a group of software developers are building a project, programmers tend to work on separate parts of the code base. By using semantic analysis on the commit message it should be possible to distinguish these two cases, but I haven't tried it here.

Instead I just used Kibana's area charts to study the correlation between "re-authorship" (when the author of the new code is the same person who wrote the older version) and the age of the code. As the age of the code can vary from minutes to years, I decided to run the visualizations in a log_2 scale (for example 0 corresponds to $2^0 = 1$ days and 10 is $2^{10} = 1024$ days or about 2.8 years). A sample area chart is shown in Figure 4. Any analysis based on plain raw histograms is a bit difficult because different repos have wildly different numbers of commits, so in Figures 5 and 6 the y-axis shows the actual percentage of re-authorship. There is a clear pattern that when a piece of code gets replaced, the younger it is the more likely it is that the new author is the same as the old one. I wonder what kinds of more sophisticated analysis is possible, for example doing more analysis on commit messages or trying to forecast buggy areas of the code base.

Figure 5: The age distribution of replaced code, with a color indication on "re-authorship" on repositories of Apache Spark, Angular.js and the git itself.

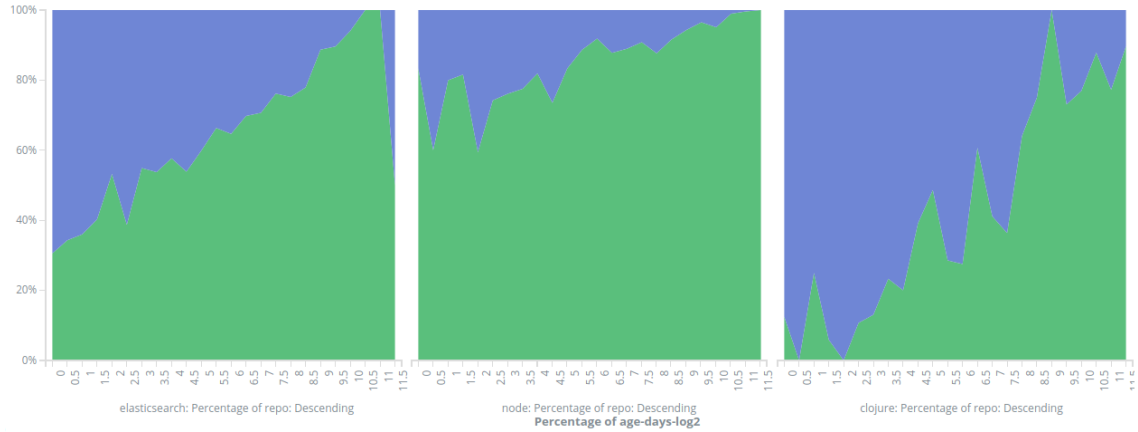

Figure 6: The age distribution of replaced code, with a color indication on "re-authorship" on repositories of Elasticsearch, Node.js and Clojure.