How to Analyze Git Repositories with Command Line Tools: We're not in Kansas Anymore

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
dds@aueb.gr

Georgios Gousios
Department of Software Technology
Delft University of Technology
Delft, The Netherlands
g.gousios@tudelft.nl

ABSTRACT

Git repositories are an important source of empirical software engineering product and process data. Running the Git command-line tool and processing its output with other Unix tools allows the incremental construction of sophisticated data processing pipelines. Git data analytics on the command-line can be systematically presented through a pattern that involves fetching, selection, processing, summarization, and reporting. For each part of the processing pipeline, we examine the tools and techniques that can be most effectively used to perform the task at hand. The presented techniques can be easily applied, first to get a feeling of version control repository data at hand and then also for extracting empirical results.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Command and control languages*;

KEYWORDS

Git; data analytics; command-line tools; pipes and filters; empirical software engineering

ACM Reference Format:
Diomidis Spinellis and Georgios Gousios. 2018. How to Analyze Git Repositories with Command Line Tools: We're not in Kansas Anymore. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3183469

1 INTRODUCTION

Git repositories are an important source of empirical software engineering product and process data [1]. Git's native user interface is the corresponding command-line tool, so knowing how to use the tool and process its output in an effective way is a key skill for front-line researchers. Applying Unix command-line tools to the output of the *git* command allows the incremental construction of sophisticated data processing pipelines. This method can be profitably employed for the rapid prototyping of software engineering data analytics tasks [4, 8]. Under this approach, the shell's read-eval-print loop (REPL) and its editing facilities are used to construct the processing pipeline incrementally by combining Git with diverse Unix built-in and add-on command-line tools.

The use of Unix command tools to analyze Git repositories offers interactivity, readability, performance, scalability, and portability. A processing pipeline is typically built interactively and verified bit by bit, until it molds into the exactly required form. This allows each step to be individually verified and fine-tuned on the actual data. In addition, powerful individual commands, such as *sort* and *awk*, and their combination as filters into pipelines [5] or directed acyclic processing graphs [9] raise the level of abstraction, thus resulting in a concise and readable expression of the intended processing. Pipelined filters also aid scalability and performance. In many cases the data flows are not constrained by the (limited) main memory size, the intermediate results need not be saved in (slow) secondary storage, and filters of multiple processing stages can be executing in parallel (on many CPU cores). Moreover, the command-line processing approach benefits from the portability of Unix, which allows the tools to run on individual workstations, on (often cloud-based) servers, and on supercomputers.

We introduce Git's storage model, because it is important for aspiring repository miners to understand the underlying snapshot graph structure and the limitations it imposes to the analysis. Git stores its data as a graph. Each node represents a revision, with edges marking the revision's parents. The contents of each node are the commit's metadata and a complete snapshot of the development tree. This means that differences between revisions and the revision log are not stored in the repository, but can be reconstructed by analyzing the graph. Tags and branches are simply names for specific nodes; branches advance to point to each new revision, while tags are permanently attached to the revision they were first associated with.

2 TOOLS AND PIPELINES

Data analytics with pipelines of Unix tools can be systematically presented through a pattern that involves fetching, selection, processing, summarization, and reporting. For example, finding the day of the week in which a system's developers perform the highest number of bug-fix commits involves *fetching* a log of commits from the Git log, *selecting* the day from commits that fix bugs, *processing* the result to group the records by day, *summarizing* the records into a count for each day, and *reporting* the counts as a chart.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3183469

As typical in repository mining tasks, we first need to select the repositories to work with based on a set of criteria [6]. For this, we use GHTorrent's [2] *ssh*-based MySQL access service and SQL queries to obtain repositories with the required characteristics [3]. The selected repositories are then cloned (in a so-called *bare* configuration) in parallel (using *xargs -P*) to a local disk.

The *git* command-line tool has a notoriously complex command-line interface. However, using just a few key commands allows researchers to access source code across revisions without requiring expensive copies to disk. Specifically, combinations of *git ls-tree*, *git grep* and *git show* allows us to stream contents from selected files and revisions to a pipe for further processing. In addition, the *git log* and *git blame* commands offer access to software development process data, regarding revisions and code authorship.

An alternative way for accessing revisions involves making them available on the filesystem namespace. This can be easily performed by checking out a particular version with *git checkout*. However, the checkout operation can require considerable processing time and secondary storage space, which can be a problem for large repositories and for analyzing time series data. Tools such as GitFS and RepoFS address this problem by allowing one to mount a Git repository as a virtual filesystem. This approach makes all revisions visible without requiring an explicit checkout operation.

After extracting the data with *git* commands, we can apply traditional Unix tools to perform relational algebra operations, without needing to store the data in an intermediate SQL database. The Unix *grep* command allows us to filter out records from further processing, while *join* can merge two streams on common key columns. We can simulate a grouping operation with *awk*, while *sort* and *uniq* allow us to efficiently create ordered and counted lists. Moreover, we can use *sed*, *cut* or any custom tool that accepts and returns text input to modify the stream in place, which is analogous to applying a user-defined function in an SQL query.

At times, using generic Unix tools to process Git data and combining them with other data sources may be cumbersome or downright impossible. Fortunately, we have several other options. Initially, we should pick the right programming language; if performance is important, or if the plan is to analyze hundreds of thousands of repositories, we should go for C++ or a modern equivalent, such as Rust or Go. In the majority of cases however, Python or Ruby offer adequate performance. All modern languages offer bindings to *libgit2*, which provides fast access to the Git object graph, independently of the *git* tools. When developing tools, we should strive to make them work in the Unix way: each of our tools should do *one thing right*, accept text at its input and return easily parsable text at its output. Operation modes and options, such as database connection strings, should be provided as command-line arguments. Finally, we should make sure that our tools work; for that, unit testing is our friend, while continuous integration provides us a safety net for accepting code contributions if our tools become successful.

3 EXPLORING AND REPORTING

A big part of empirical work in software engineering is exploratory. At this stage, researchers try to understand the data, perform basic hypothesis testing, or draw simple plots of raw data. While the time-honoured recipe of taking notes on physical notebooks is still valuable, fortunately we have better tools in our hands. *Jupyter* notebooks allow us to perform, document and share exploratory analysis tasks in a literate programming style. *Jupyter* allows us to seamlessly combine Unix pipelines with Python libraries for plotting or statistical analysis.

The final step of all research efforts is reporting the results. Here we advocate the adoption of a notebook-like style [7], where commands are embedded within a research paper's markup. This practice records the provenance of the provided figures, promotes repeatability, and avoids embarrassing mistakes.

4 CONCLUSION

The use of Git desktop GUI applications may be sufficient for beginner students and developers, but it deprives researchers the affordances required for many exploratory software engineering data analytics tasks. Although the outlined techniques are not widely known, they can be easily applied, first to get a feeling of the Git data at hand and then also for extracting empirical results. This allows a broad section of the software engineering community to benefit from them. Specifically, software engineering researchers applying the presented techniques will profit from the flexibility, versatility, scalability, power, robustness, and efficiency offered by Git and other command-line tools and data processing pipelines.

Acknowledgements

The project associated with this work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732223. This work has been partially funded by the GSRT Research Support programme 2016–2017.

REFERENCES

[1] 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. In *MSR'09: 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSR.2009.5069475

[2] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: Github's Data from a Firehose. In *9th IEEE Working Conference on Mining Software Repositories (MSR)*, Michele Lanza, Massimiliano Di Penta, and Tao Xie (Eds.). IEEE, 12–21. https://doi.org/10.1109/MSR.2012.6224294

[3] Georgios Gousios and Diomidis Spinellis. 2017. Mining Software Engineering Data from GitHub. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 501–502. https://doi.org/10.1109/ICSE-C.2017.164 Technical Briefing.

[4] Brian W.. Kernighan. 2008. Sometimes the Old Ways Are Best. *IEEE Software* 25, 6 (Nov 2008), 18–19. https://doi.org/10.1109/MS.2008.161

[5] Regine Meunier. 1995. The Pipes and Filters Architecture. In *Pattern Languages of Program Design*, James O. Coplien and Douglas C. Schmidt (Eds.). Addison-Wesley, Reading, MA, Chapter 22, 427–440.

[6] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for Engineered Software Projects. *Empirical Software Engineering* 22, 6 (01 Dec 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[7] Helen Shen. 2014. Interactive Notebooks: Sharing the Code. *Nature* 515, 7525 (2014), 151–152. https://doi.org/10.1038/515151a

[8] Diomidis Spinellis. 2015. Tools and Techniques for Analyzing Product and Process Data. In *The Art and Science of Analyzing Software Data*, Tim Menzies, Christian Bird, and Thomas Zimmermann (Eds.). Morgan-Kaufmann, 161–212. https://doi.org/10.1016/B978-0-12-411519-4.00007-0

[9] Diomidis Spinellis and Marios Fragkoulis. 2017. Extending Unix Pipelines to DAGs. *IEEE Trans. Comput.* 66, 9 (Sept. 2017), 1547–1561. https://doi.org/10.1109/TC.2017.2695447