VCS Implementations and Comparison

Abhinav Patel

October 31, 2017

Part 1: Introduction

This post was inspired by the lecture regarding git basics. It will aim to take an in depth look into how git is actually implemented, some of its key features, and compare it with other commonly used version control systems in the data science and computer science fields.

Part 2: Git Internals

Git is a *version control software* (VCS), which can be used to do anything from data recovery to feature and deploy management for complex components that are maintained by a programming entity. It was originally defined to be a toolkit that enables low-level work rather than a user-friendly framework, and stores its internals within the `.git/` directory. This directory typically has the following structure when intialized:

```
> ls -l ./.git/
  HEAD
  branches/
  config/
  description
  hooks/
  index
  info/
  objects/
  refs/
```

As expected, the `config`, `info`, `description`, and `branches` directories, contain details regarding that specific repository's configuration, general information (as well as files to ignore), and working branches. The `hooks` directory contains configurations for a type of client-server script. The `refs` directory contains references to commits, and `HEAD` points to the current branch checked out. This leaves two remaining directories: `objects`, which stores all content in the git database, and `index`, where staging information is stored.

Now that the basics of git implementation have been laid out, lets take a look at what all this is based around: *the git object*.

Git Objects

At the end of the day, git is simply an addressable file system. This is simply a key-value store, similar to JSON, NoSQL, a Map/Dictionary and literally thousands of other instances in the programming world. Search a key, get an object back (if it exists). Insert an object, get its key back. Here is an example of how to store an object:

```
> echo 'store this' | git hash-object -w --stdin
  d6704760b4b4aece5915caf5c684aece5fe3e4
```

The hash returned is the search key of the new git object created. Here is how to search via search key:

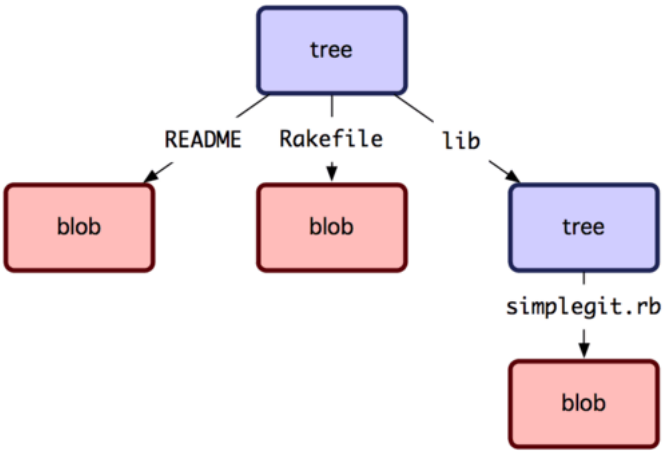
```
> git cat-file -p d6704760b4b4aece5915caf5c684aece5fe3e4
  store this
```

Each SHA-1 key (search key) usually maps to a *blob object*, which can be thought of as a snapshot of the files in that specific commit.

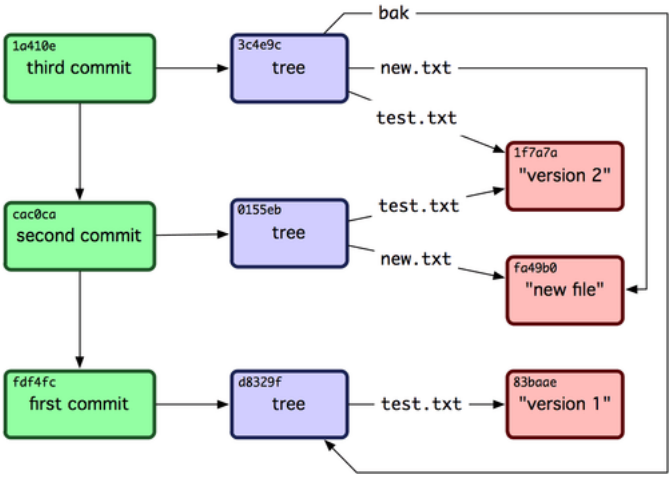
There is another type of important object in git: *the tree object*. This object, like its name suggests, follows a tree structure, which enables us to store groups of files together. All content within git is stored as trees and blobs. Trees correspond to UNIX directories, and blobs represent file contents and commit differences. One or more tree entries are contained within a single tree object, and each node contains either a SHA-1 pointer to a blob (leaf node) or a child subtree. For example, the tree of a master branch may look something like:

```
> git cat-file -p master^{tree}
  100644 blob d6704760b4b4aece5915caf5c684aece5fe3e4    README
  100644 blob 0182275347e9743ac307ebf1336a2793aad558e1  Rakefile
  040000 tree 15bd5df4ac493d9098242509a6db5827b6e28eb8  lib
```

Conceptually, a tree object can resemble:



But where do we get information about who saved each blob object, how do we efficiently track differences between snapshots optimally, and how do we store other information pertaining to a commit (like a message)? The answer is a *commit object*. A commit object specifies the top-level tree for the snapshot of the project at that point (the user, email and timestamp, etc). So in reality, the tree structure for an entire branch within our repository will resemble something like:



We now have a basic picture as to how the internals of git actually work and the abstractions for how storage and implementation happens.

Part 3: Comparison of Different VCS

We just discussed the internals of the most common version control system: git. But git is just one version control system, with its own pros and cons. How does git compare against other powerful VCS software, specifically CVS, SVN and Mercurial? Let's look at the capabilities and differences between the four VCS, which are by far the tree most dominant, and see what pitfalls and strengths they have.

CVS

CVS (Concurrent Versions System) is a rather old server-based system, and has been around since the 1980's. It holds and is released under a GNU license, allowing users to checkout code they are working on and check in updates. It was a first-come, first serve based system, only allowing the latest version of their code to be worked on and updated. The largest pro with CVS is the fact that it has been in use for almost 40 years and is relatively robust. The major cons with it are its inflexibility when it comes to refactors, security risks, and no robust support for long term branching.

SVN

SVN stands for Subversion, which was created by the Apache Software Foundation, which is an extremely reputable and well known organization responsible for widely used software such as Hadoop, Tomcat, and Spark. It was originally developed as a way to fix vulnerabilities with CVS while maintaining compabitibility. It introduced the concept of *atomic operations* meaning all changes are applied or none are applied. Additionally, unlike CVS, SVN has efficient support of fork, branch, and merge operations. The pros with SVN are that it allows atomic operations, cheaper tree operations, widely available and developed, and does not follow a peer-to-peer (P2P) model. The cons are that it has a relatively slow overall process speed, and is somewhat lacking in the number of available repository commands.

Git

Git takes a drastically approach to version control as compared to CVS and SVN. It follows a distributed, decentralized model, and runs drastically faster. The pros of git are its usability, plethora of supported operations, cheap tree operations, full revision history racking, and drastically faster runtime. Really, the only two cons with git are that there is a high learning curve, and a single developer working on his/her own will not end up utilizing most of the available features.

Mercurial

Mercurial is another decentralized version control system. As opposed to git and SVN, Mercurial is implemented in primarily Python rather than C. Due to this, Python developers sometimes use Mercurial since it would allow easier access to checking out changes and branch operations. It has many similarities with SVN in terms of how to use it, but it is not scriptable and is usually used as an extension. The pros of Mercurial are are that it is easier to use than git, has better documentation, and is still decentralized. Some major cons are that two parents cannot be merged and how it is not scriptable.

Sources

https://biz30.timedoctor.com/git-mecurial-and-cvs-comparison-of-svn-software/
https://git-scm.com/book/en/v1/Git-Internals
https://github.com/pluralsight/git-internals-pdf
https://www.mercurial-scm.org/wiki/DeveloperInfo
http://www.gnu.org/software/libcvs-spec/guide/internals/architecture.html
https://stackoverflow.com/questions/2332833/how-exactly-does-subversion-store-files-in-the-repository
https://www.g2crowd.com/categories/version-control-systems