Git

Version control systems (VCSs) - tools used to track changes to source code - help maintain a history of changes - facilitate collaboration. - track changes to a folder and its contents in a series of snapshots, where each snapshot encapsulates the entire state of files/folders within a top-level directory. - store metadata: who created each snapshot, messages associated with each snapshot

Modern VCSs let you easily answer questions like:

- Who wrote this module?
- When was this particular line of this particular file edited? By whom? Why was it edited?
- Over the last 1000 revisions, when/why did a particular unit test stop working?

While other VCSs exist, **Git** is the de facto standard for version control. This XKCD comic captures Git's reputation:

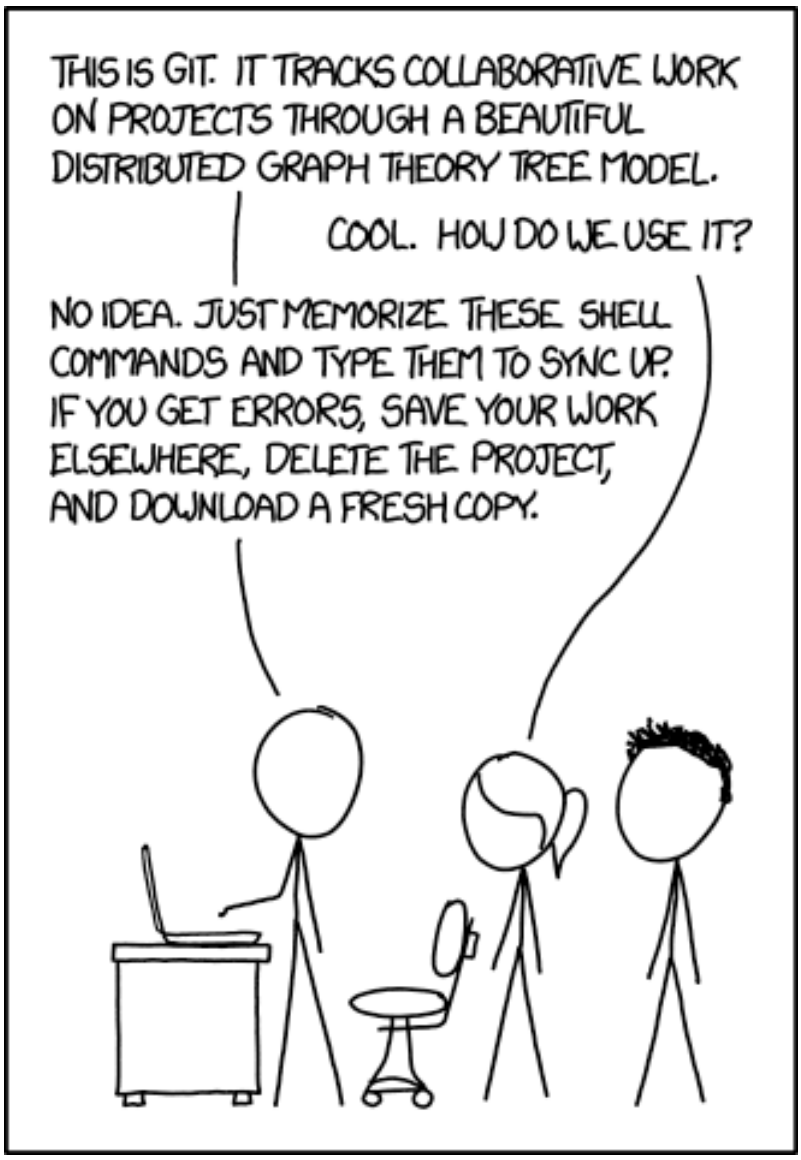

Figure 1: xkcd 1597

While Git admittedly has an ugly interface, its underlying design and ideas are beautiful. While an ugly interface has to be *memorized*, a beautiful design can be *understood*. For this reason, we give a bottom-up explanation of Git, starting with its data model and later covering the command-line interface. Once the data model is understood, the commands can be better understood, in terms of how they manipulate the underlying data model.

Git's data model

Git has a well thought-out model that enables all the nice features of version control, like maintaining history, supporting branches, and enabling collaboration.

Snapshots

Git models the history of a collection of files and folders within some top-level directory as a series of snapshots. In Git terminology, a file is called a *"blob"*, and it's just a bunch of bytes. A directory is called a *"tree"*, and it maps names to blobs or trees (so directories can contain other directories). A snapshot is the top-level tree that is being tracked. For example, we might have a tree as follows:

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

The top-level tree contains two elements, a tree "foo" (that itself contains one element, a blob "bar.txt"), and a blob "baz.txt".

Modeling history: relating snapshots

How should a version control system relate snapshots? One simple model would be to have a linear history. A history would be a list of snapshots in time-order. For many reasons, Git doesn't use a simple model like this.

In Git, a history is a *directed acyclic graph* (DAG) of snapshots. All this means is that each snapshot in Git refers to a set of "parents", the snapshots that preceded it. It's a set of parents rather than a single parent (as would be the case in a linear history) because a snapshot might descend from multiple parents, for example due to combining (merging) two parallel branches of development.

Git calls these snapshots "commit"s. Visualizing a commit history might look something like this:

```
o <-- o <-- o <-- o
            ^
             \
              --- o <-- o
```

In the ASCII art above, the os correspond to individual commits (snapshots). The arrows point to the parent of each commit (it's a "comes before" relation, not "comes after"). After the third commit, the

history branches into two separate branches. This might correspond to, for example, two separate features being developed in parallel, independently from each other. In the future, kkthese branches may be merged to create a new snapshot that incorporates both of the features, producing a new history that looks like this, with the newly created merge commit shown in bold:

```
o <-- o <-- o <-- o <---- <strong>o</strong>
            ^            /
             \          v
              --- o <-- o
```

Commits in Git are immutable. This doesn't mean that mistakes can't be corrected, however; it's just that "edits" to the commit history are actually creating entirely new commits, and references (see below) are updated to point to the new ones.

Data model, as pseudocode

Git's data model written down in pseudocode:

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | file>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

It's a clean, simple model of history.

Objects and content-addressing

An "object" is a blob, tree, or commit:

```
type object = blob | tree | commit
```

In Git data store, all objects are content-addressed by their SHA-1 hash.

```
objects = map<string, object>

def store(object):
    id = sha1(object)
```

```
    objects[id] = object

def load(id):
    return objects[id]
```

Blobs, trees, and commits are unified in this way: they are all objects. When they reference other objects, they don't actually *contain* them in their on-disk representation, but have a reference to them by their hash.

For example, the tree for the example directory structure above (visualized using `git cat-file -p 698281bc680d1995c5f4caaf3359721a5a58d48d`), looks like this:

```
100644 blob 4448adbf7ecd394f42ae135bbeed9676e894af85    baz.txt
040000 tree c68d233a33c5c06e0340e4c224f0afca87c8ce87    foo
```

The tree itself contains pointers to its contents, `baz.txt` (a blob) and `foo` (a tree). If we look at the contents addressed by the hash corresponding to baz.txt with `git cat-file -p 4448adbf7ecd394f42ae135bbeed9676e894af85`, we get the following:

```
git is wonderful
```

References

Now, all snapshots can be identified by their SHA-1 hash. That's inconvenient, because humans aren't good at remembering strings of 40 hexadecimal characters.

Git's solution to this problem is human-readable names for SHA-1 hashes, called "references". References are pointers to commits. Unlike objects, which are immutable, references are mutable (can be updated to point to a new commit). For example, the `master` reference usually points to the latest commit in the main branch of development.

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

With this, Git can use human-readable names like “master” to refer to a particular snapshot in the history, instead of a long hexadecimal string.

One detail is that we often want a notion of “where we currently are” in the history, so that when we take a new snapshot, we know what it is relative to (how we set the `parents` field of the commit). In Git, that “where we currently are” is a special reference called “HEAD”.

Repositories

Finally, we can define what (roughly) is a Git *repository*: it is the data `objects` and `references`.

On disk, all Git stores is objects and references: that’s all there is to Git’s data model. All `git` commands map to some manipulation of the commit DAG by adding objects and adding/updating references.

Whenever you’re typing in any command, think about what manipulation the command is making to the underlying graph data structure. Conversely, if you’re trying to make a particular kind of change to the commit DAG, e.g. “discard uncommitted changes and make the ‘master’ ref point to commit `5d83f9e`”, there’s probably a command to do it (e.g. in this case, `git checkout master`; `git reset --hard 5d83f9e`).

Staging area

This is another concept that’s orthogonal to the data model, but it’s a part of the interface to create commits.

One way you might imagine implementing snapshotting as described above is have a “create snapshot” command that creates a new snapshot based on the *current state* of the working directory. Some version control tools work like this, but not Git. We want clean snapshots, and it might not always be ideal to make a snapshot from the current state. For example, imagine a scenario where you’ve implemented two separate features, and you want to create two separate commits, where the first introduces the first feature, and the next introduces the second feature. Or imagine a scenario where you have debugging print statements added all over your code, along with a bugfix; you want to commit the bugfix while discarding all the print statements.

Git accommodates such scenarios by allowing you to specify which modifications should be included in the next snapshot through a mechanism called the “staging area”.

Git command-line interface

To avoid duplicating information, we're not going to explain the commands below in detail. See the highly recommended Pro Git for more information, or watch the lecture video.

Basics

The `git init` command initializes a new Git repository, with repository metadata being stored in the `.git` directory:

```
$ mkdir myproject
$ cd myproject
$ git init
Initialized empty Git repository in /home/missing-semester/myproject/.
    git/
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

How do we interpret this output? "No commits yet" basically means our version history is empty. Let's fix that.

```
$ echo "hello, git" > hello.txt
$ git add hello.txt
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   hello.txt

$ git commit -m 'Initial commit'
[master (root-commit) 4515d17] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 hello.txt
```

With this, we've `git add`ed a file to the staging area, and then `git commit`ed that change, adding a simple commit message "Initial commit". If we didn't specify a `-m` option, Git would open our text editor to allow us type a commit message.

Now that we have a non-empty version history, we can visualize the history. Visualizing the history as

a DAG can be especially helpful in understanding the current status of the repo and connecting it with your understanding of the Git data model.

The `git log` command visualizes history. By default, it shows a flattened version, which hides the graph structure. If you use a command like `git log --all --graph --decorate`, it will show you the full version history of the repository, visualized in graph form.

```
$ git log --all --graph --decorate
* commit 4515d17a167bdef0a91ee7d50d75b12c9c2652aa (HEAD -> master)
  Author: Missing Semester <missing-semester@mit.edu>
  Date:   Tue Jan 21 22:18:36 2020 -0500

      Initial commit
```

This doesn't look all that graph-like, because it only contains a single node. Let's make some more changes, author a new commit, and visualize the history once more.

```
$ echo "another line" >> hello.txt
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
     directory)

        modified:   hello.txt

no changes added to commit (use "git add" and/or "git commit -a")
$ git add hello.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   hello.txt

$ git commit -m 'Add a line'
[master 35f60a8] Add a line
 1 file changed, 1 insertion(+)
```

Now, if we visualize the history again, we'll see some of the graph structure:

```
* commit 35f60a825be0106036dd2fbc7657598eb7b04c67 (HEAD -> master)
| Author: Missing Semester <missing-semester@mit.edu>
| Date:   Tue Jan 21 22:26:20 2020 -0500
|
|     Add a line
|
* commit 4515d17a167bdef0a91ee7d50d75b12c9c2652aa
  Author: Anish Athalye <me@anishathalye.com>
```

```
  Date:   Tue Jan 21 22:18:36 2020 -0500

      Initial commit
```

Also, note that it shows the current HEAD, along with the current branch (master).

We can look at old versions using the `git checkout` command.

```
$ git checkout 4515d17  # previous commit hash; yours will be different
Note: checking out '4515d17'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 4515d17 Initial commit
$ cat hello.txt
hello, git
$ git checkout master
Previous HEAD position was 4515d17 Initial commit
Switched to branch 'master'
$ cat hello.txt
hello, git
another line
```

Git can show you how files have evolved (differences, or diffs) using the `git diff` command:

```
$ git diff 4515d17 hello.txt
diff --git c/hello.txt w/hello.txt
index 94bab17..f0013b2 100644
--- c/hello.txt
+++ w/hello.txt
@@ -1 +1,2 @@
 hello, git
 +another line
```

- `git help <command>`: get help for a git command
- `git init`: creates a new git repo, with data stored in the `.git` directory
- `git status`: tells you what's going on
- `git add <filename>`: adds files to staging area

- `git commit`: creates a new commit
  - Write good commit messages!
- `git log`: shows a flattened log of history
- `git log --all --graph --decorate`: visualizes history as a DAG
- `git diff <filename>`: show differences since the last commit
- `git diff <revision> <filename>`: shows differences in a file between snapshots
- `git checkout <revision>`: updates HEAD and current branch

Branching and merging

Branching allows you to "fork" version history. It can be helpful for working on independent features or bug fixes in parallel. The `git branch` command can be used to create new branches; `git checkout -b <branch name>` creates and branch and checks it out.

Merging is the opposite of branching: it allows you to combine forked version histories, e.g. merging a feature branch back into master. The `git merge` command is used for merging.

- `git branch`: shows branches
- `git branch <name>`: creates a branch
- `git checkout -b <name>`: creates a branch and switches to it
  - same as `git branch <name>; git checkout <name>`
- `git merge <revision>`: merges into current branch
- `git mergetool`: use a fancy tool to help resolve merge conflicts
- `git rebase`: rebase set of patches onto a new base

Remotes

- `git remote`: list remotes
- `git remote add <name> <url>`: add a remote
- `git push <remote> <local branch>:<remote branch>`: send objects to remote, and update remote reference
- `git branch --set-upstream-to=<remote>/<remote branch>`: set up correspondence between local and remote branch
- `git fetch`: retrieve objects/references from a remote
- `git pull`: same as `git fetch; git merge`
- `git clone`: download repository from remote

Undo

- `git commit --amend`: edit a commit's contents/message
- `git reset HEAD <file>`: unstage a file
- `git checkout -- <file>`: discard changes
- `git reset`: moves branch reference backwards in time to an older commit
  - moves a branch backwards in time as if the commit had never taken place
  - `git reset HEAD~1`: moves `HEAD` back by one commit
- `git revert`: for reversing changes already propagated to the remote. Adds a new commit of changes

Relative Refs

Relative refs are useful to refer to commits without having to type out the hash - `^`: move upwards by one commit - `^<num>`: move upwards to parent reference e.g. for a merge commit with multiple parents (default: 1) - `~<num>`: move upwards by num commits - e.g. - `git checkout master^`: checks out first parent of master - `master^ ^`: grandparent of `master` - `HEAD~4`: 4th ancestor of `HEAD` - chaining e.g. `git checkout HEAD~^2~2`: move to parent commit, move to second co-parent of commit, move two commits up - Applications - Branch forcing: directly reassign a branch to a commit - `git branch -f master HEAD~3` forces master branch to 3 parents behind `HEAD`

Moving work around

- `git cherry-pick <commit1> <commit2> <...>`: copies a series of commits below current location, i.e. add new commit corresponding to `commit1`, then `commit2`, etc.
- `git rebase -i` interactive rebase. Opens a text editor showing which commits are about to be copied below the rebase target. You can then
  - reorder commits by changing order
  - omit commits by setting `pick` off
  - squash commits (i.e. combine multiple commits)

Finding your way around

- `git tag <name>` to apply `name` to a specific commit e.g. version number
- `git describe` describes where you are relative to nearest anchor (tagged commit)

  - output: `<tag>_<numCommits>_g<hash>`
    * `tag`: tag of nearest anchor
    * `numCommits`: number of commits away to anchor
    * `hash`: hash of commit being described (not the anchor)

Advanced Git

- `git config`: Git is highly customizable
- `git clone --shallow`: clone without entire version history
- `git add -p`: interactive staging
- `git rebase -i`: interactive rebasing
- `git blame`: show who last edited which line
- `git stash`: temporarily remove modifications to working directory
- `git bisect`: binary search history (e.g. for regressions)
- `.gitignore`: specify intentionally untracked files to ignore

Miscellaneous

- **GUIs**: There are many GUI clients out there for Git. We personally don't use them and use the command-line interface instead.
- **Shell integration**: It's super handy to have a Git status as part of your shell prompt (zsh,). Often included in frameworks like Oh My Zsh.
- **Editor integration**: Similarly to the above, handy integrations with many features. fugitive.vim is the standard one for Vim.
- **Workflows**: we taught you the data model, plus some basic commands; we didn't tell you what practices to follow when working on big projects (and there are many different approaches).
- **GitHub**: Git is not GitHub. GitHub has a specific way of contributing code to other projects, called pull requests.
- **Other Git providers**: GitHub is not special: there are many Git repository hosts, like GitLab and BitBucket.

Resources

- Pro Git is **highly recommended reading**. Going through Chapters 1–5 should teach you most of what you need to use Git proficiently, now that you understand the data model. The later

chapters have some interesting, advanced material.
- Oh Shit, Git!?! is a short guide on how to recover from some common Git mistakes.
- Git for Computer Scientists is a short explanation of Git's data model, with less pseudocode and more fancy diagrams than these lecture notes.
- Git from the Bottom Up is a detailed explanation of Git's implementation details beyond just the data model, for the curious.
- How to explain git in simple words
- Learn Git Branching is a browser-based game that teaches you Git.

Exercises

1. If you don't have any past experience with Git, either try reading the first couple chapters of Pro Git or go through a tutorial like Learn Git Branching. As you're working through it, relate Git commands to the data model.
2. Clone the repository for the class website. `git clone https://github.com/missing-semester/missing-semester`
    1. Explore the version history by visualizing it as a graph. `git log --graph`
    2. Who was the last person to modify `README.md`? (Hint: use `git log` with an argument) `git log -n 1 README.md`
    3. What was the commit message associated with the last modification to the `collections:` line of `_config.yml`? (Hint: use `git blame` and `git show`)

```
git blame -L /collections:/ _config.yml
git show a88b4eac
```

3. One common mistake when learning Git is to commit large files that should not be managed by Git or adding sensitive information. Try adding a file to a repository, making some commits and then deleting that file from history (you may want to look at this).
4. Clone some repository from GitHub, and modify one of its existing files. What happens when you do `git stash`? What do you see when running `git log --all --oneline`? Run `git stash pop` to undo what you did with `git stash`. In what scenario might this be useful?
5. Like many command line tools, Git provides a configuration file (or dotfile) called `~/.gitconfig`. Create an alias in `~/.gitconfig` so that when you run `git graph`, you get the output of `git log --all --graph --decorate --oneline`.
6. You can define global ignore patterns in `~/.gitignore_global` after running `git config --global core.excludesfile ~/.gitignore_global`. Do this, and set up your global gitignore file to ignore OS-specific or editor-specific temporary files, like `.DS_Store`.

7. Clone the repository for the class website, find a typo or some other improvement you can make, and submit a pull request on GitHub.