# git - the stupid content tracker

## git eats trees.

Henry S. Grasshorn Gebhardt

September 9, 2025

# Outline

# Version Control? Version Control.

Version control helps to:

- Save history.
- Keep track of changes.
- Merge code.
- Share code. (Don't be a git!)
- Consistency checking, e.g., when running code or configs elsewhere.
- ...

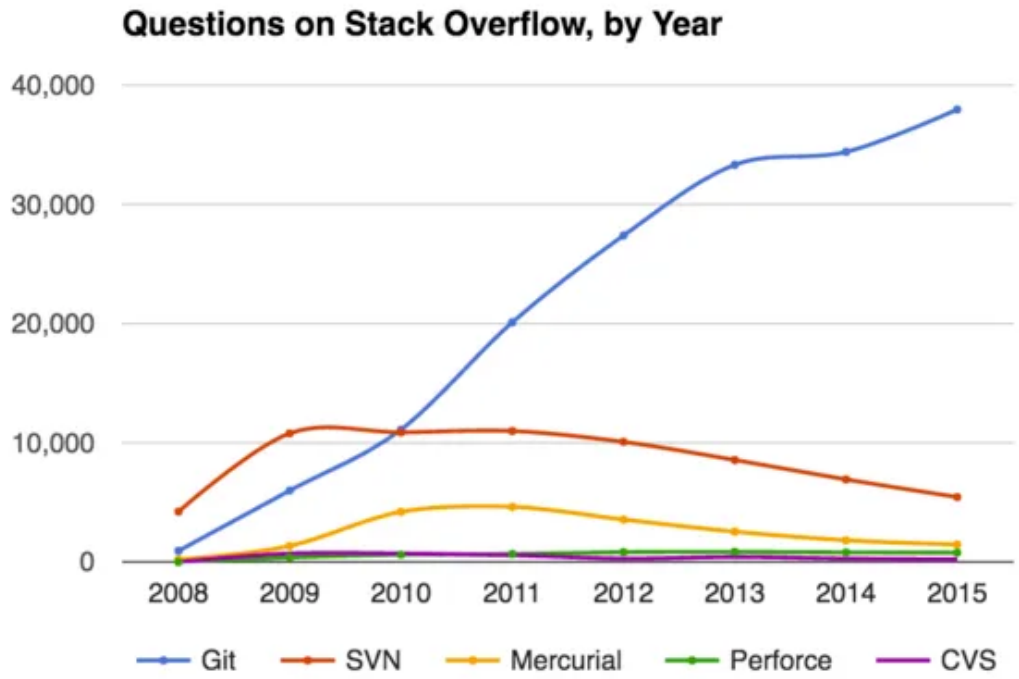
Git, Mercurial, ~~Bazaar~~, ~~SVN (why bother?)~~, ~~CVS~~, ~~Monotone~~, ~~DARCS~~, ...
"Theory of Patches"

# Git was created by Linus Torvalds for Linux kernel development

Linux developers used to use `BitKeeper` until it wasn't free anymore.

So Linus Torvalds created git, which always had a reputation for being too complicated to use. First release 7 April 2005.

There are *plumbing* commands and *porcelain* commands.



# What is a patch?

Patches are text files that show changes.

## Create a patch:

```
$ diff -Naur file1 file2 > changes.patch
```

```
henry@FrappleDapple-10 SPHEREx-L4-Cosmology-Pipeline % diff -Naur --color=always makefile makefile.new
--- makefile    2025-09-08 14:43:53
+++ makefile.new        2025-09-08 14:43:33
@@ -44,7 +44,7 @@
 FORMATTER = ./scripts/black-formatting.sh

 # See https://github.com/conda/conda-build/issues/4251#issuecomment-1053460542
-PIP_INSTALL = conda run -n chimera pip install --editable
+PIP_INSTALL = conda run -n chimera pip install --no-build-isolation --no-deps --editable

 GENERATE_CONDA_LOCK = cd "$(shell dirname "$(1)")"; conda-lock -f "$(shell basename "$(2)")" -p osx-arm64 -p linux-64

henry@FrappleDapple-10 SPHEREx-L4-Cosmology-Pipeline %
```

## Apply a patch:

```
$ patch -p1 < changes.patch
```

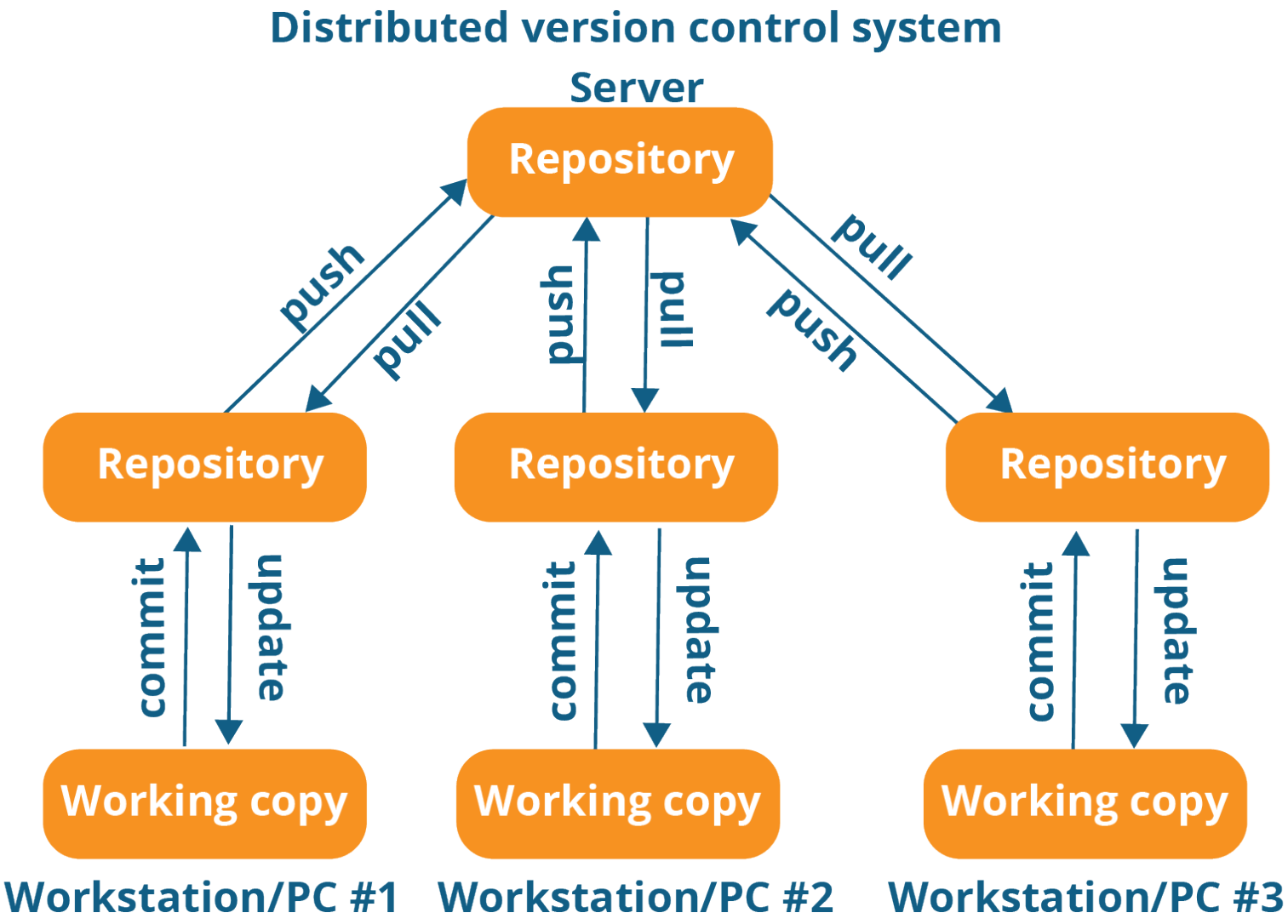
# Git History

`https://git-scm.com`

History is a DAG (directed acyclic graph). *Explain graph.*

Distributed, not centralized. *Every clone has the full history.*

# Distributed: Each git clone is a full repository




`git pull` = `git fetch` + `git merge`

# Git doesn't know about directories. . . whaaat?

Git only knows content. (blobs)

How that content is assembled. (trees)

And history. (commits)

Each commit stores the full source code (no patches!).

# blobs, trees, and commits are identified by their SHA1-sum

9fc058eae64b097d7a270a404527375526e65cea

A hash is a (hopefully) unique number to identify some information, like a file.

SHA-1 is a 160-bit number. It happens to be cryptographically secure.

Blobs, trees, and commits are identified by their SHA1 sum.

$\Rightarrow$ Efficient de-duplication and compression

# Terminology: blah, blah, blah,...

WORKDIR (directory where files are checked out)
GITDIR (`.git`)
HEAD (currently checked out commit)
Index (staging area for next commit)
Local repository
Upstream repository
Stash

# Per-seat Initialization

Initialization once per machine (`~/.gitconfig`):

```
$ git config --global user.name "Henry VIII"
$ git config --global user.email "h@here.com"
```

Set your `EDITOR` variable in `~/.bashrc` or `~/.profile` or wherever you set your environment variables.

# Making history

First, add changes to staging area (the *index*):

```
git add <file>  # Add your changes to the index.
git add -p      # Be selective about what to add.
```

Then, commit:

```
git commit  # Commits your changes.
```

Add a short title, and then a longer description, e.g.,

```
commit b56f36b4f7e7559ada975b48c3fd4aa27d3de672 (HEAD -> main)
Author: Henry Gebhardt <hsggebhardt@fastmail.com>
Date:   Tue Sep 9 00:11:13 2025 -0700

    talk: cleanup duplicate slides variations

    There were several incantations of several slides. Through an ingenious
    trick that involved looking at the source code, the culprit was found.
```

# What to commit

- The bare minimum to recreate the project.
- Plots! (They are the minimum to recompile the LaTeX.)

`.gitignore` can appear anywhere in the git repository, and contains files to ignore, e.g.,

.gitignore

```
*.aux
*.log
*.toc
```

Reasons: avoid conflicts, save space.

# Pull Requests (PR) and merging

PRs are used to coordinate work between multiple people.

Merge commit: Creates a new commit, keeping both histories.

Squash commit: Squashes branch history into a single commit.

Rebase: Takes branch and replays it ontop of base branch.

Fast-forward: No new commit, just advance the base branch.

Forced push: Overwrite what's on the remote.

Best kind of merge is a fast-forward: no merging, no conflicts!

`https://github.com/SPHEREx/SPHEREx-L4-Cosmology-Pipeline/blob/main/doc/PRPolicy.md#merging-choices`

# Merging branches and conflicts

```
git config --global merge.tool nvimdiff
git mergetool --tool-help
```

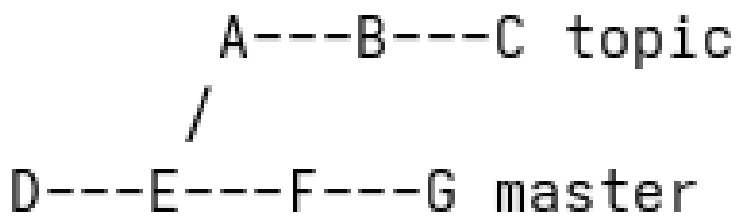
```
git merge <branches>...
git mergetool
```

Four files:

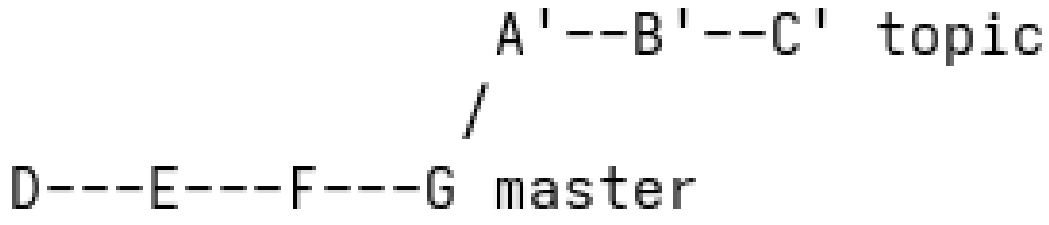
- LOCAL: Current version.
- BASE: Last common version.
- REMOTE: To-be-merged-in version.
- MERGED: The version we want.

(Demonstration: Add README.)

# Rebasing




```
git checkout topic
git rebase -i master
```




```
git push --force
```

# Hosting your git repository

Providers:

Github: `github.com`

Gitlab: `gitlab.com`

Sourcehut: `sourcehut.org`

...

Your own:

SSH server: Your own workstation/server.

```
$ mkdir -p ~/repos/newawesomeproject.git
$ cd ~/repos/newawesomeproject.git
$ git init --bare
```

SSH server: `http://gitolite.com/` (Fine-grained permissions)

Forgejo: `https://forgejo.org/` (Full-on website)

# git cheat sheet

Here's the *porcelain*:

`https://about.gitlab.com/images/press/git-cheat-sheet.pdf`

```
man gittutorial
man git-reflog
```

Initialization once per machine:

Create the file `~/.gitconfig`.

Set your `EDITOR` variable in `~/.bashrc` or `~/.profile`.

# Appendix

# git cheat sheet

Here's the *porcelain*:

`https://about.gitlab.com/images/press/git-cheat-sheet.pdf`

# git stores its information in a .git directory

```
newrepo/
    .git/
    .gitignore
    README.md
    doc/
    src/
    test/
```

# git help command

Useful commands:

`git status` Where am I?

`git diff` What did I just do?

`git diff --staged` What will I do?

`git log` What have I done?

`gitk --all` Let's climb trees!

`git describe --always --tags --dirty` Who am I?

# Pushing and pulling

```
$ git push <remote> <localbranch>:<remotebranch>
$ git push --set-upstream
$ git pull
$ git remote -v
```

# Trees, yum!

## git eats trees. . . nom,nom

Branches are cheap!

`git branch -a` List branches.

`git checkout [-b <newbranch>] <starthere>` Checkout and make a new branch.

`git merge <otherbranches>...` Trees eating trees!

`git merge --squash <otherbranches>...` Squash trees!

`git rebase -i <branchname>` Clean up your history!

# Initial checkout

Existing repository:

```
$ git clone <url>
```

New repository:

```
$ mkdir newrepo
$ cd newrepo
$ git init
```

# Sending and receiving patches

`git format-patch` Create a patch

`git send-email` Send an entire set of patches as emails.

`git am, git apply` Apply other people's patches.

# Where to commit: Branches

*master* (or *main*): Main development branch.

release branches: More stable branches that you might want to keep supporting by fixing bugs and cherry-picking commits from *master*.

feature branches: Development branch for specific features. By convention, they often start with your initials, e.g., *hg/integration_method_B*. Should eventually be merged into *master*.

# Play nice together!

```
$ git svn
```

Works by calling “`git fast-import`”.

# Upstreams and tracking branches

A branch can be set to track an upstream:

`git push -u origin <localbranch>:<remotebranch>`

(or, equivalently, use `--set-upstream`)

# Ah, I did something stupid...

Recovery might be possible by looking into `.git/logs/`.

`git reflog` parses it for you.

# Other commands

Graphs: `git log --graph`

More graphs: `gitk --all`

Tags: `git tag`

Hooks: `man githooks`; `cd .git/hooks/`

Submodules: `git submodule`

Rewrite history: `git filter-branch`

Collect garbage: `git gc`