Version Control

Ramses van Zon

PHY1610 Winter 2025

What is version control?

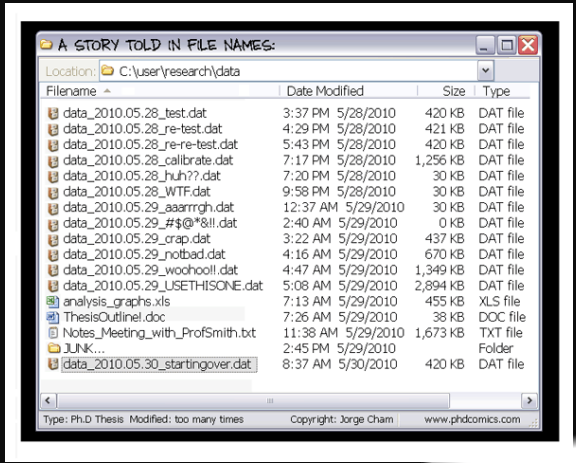
- Version control is a tool for **managing changes** in a set of **files**.
- Keeps historical versions.

Why use it?

- Makes **collaborating** on code easier/possible/less violent.
- Helps you **stay organized**.
- Allows you to **track changes** in the code.
- Allows **reproducibility** in the code.
- Allows to maintain multiple versions in **branches**

That is not a system I would recommend $\longrightarrow$

I guess this is also a version control system:



src: PhD Comics

Types of version control workflows

Centralized

- One authoratitive, central remote repository
- Local clones that can check in changes

Examples:

CVS, Subversion (SVN), GitHub, GitLab

Distributed

- Every clone is a valid, complete repository
- Can clone any repo
- Can pull from any other and resolve differences

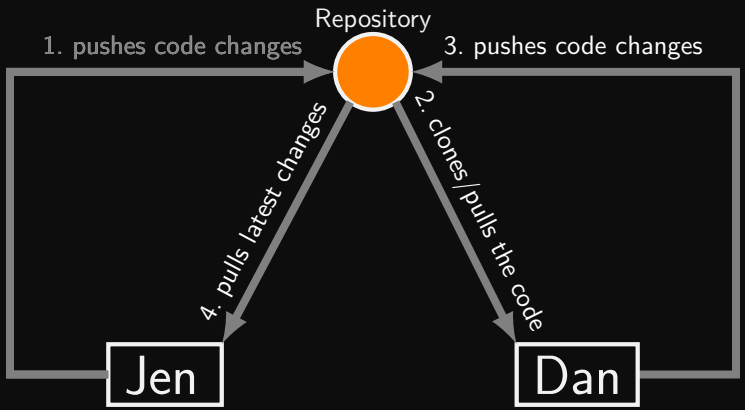
Examples: Git, Mercurial

Seems great and flexible. But people like a central repo for their sanity and for publishing code.

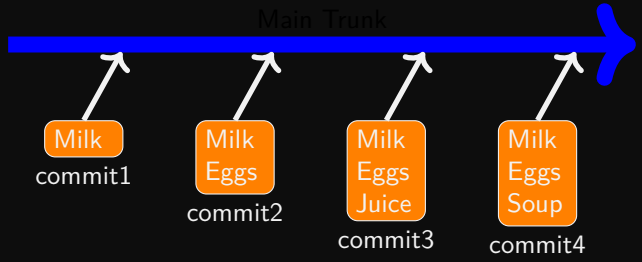
In common

- They are equivalent when you're only working with one, local, repository
- One must be explicit in what file changes are tracked and when versions are commited.
- One can ask for history, and go backwards (and forwards) in time

How does (central) version control work?



Basic Change Commits (Local)

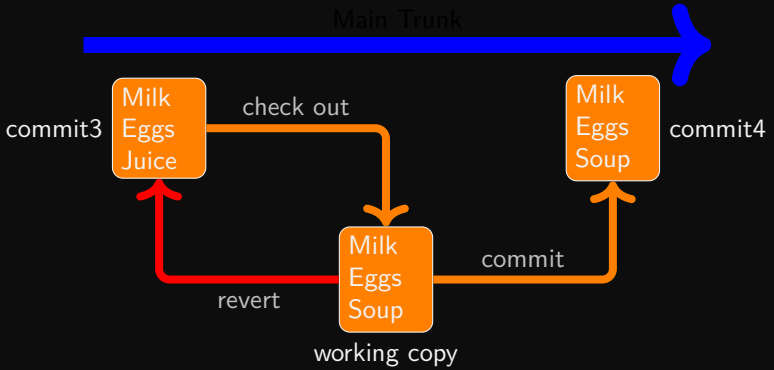


Commits store ***differences*** in the file.

Files that are not changed are not stored again.

Yet commits act as snapshots when you check them out.

Checkout and edit



You do not have direct access to the repo.

Instead, you checkout a working copy.

Once you are happy with your changes in the working copy, you prepare stage and then apply the commit.

Version control: git

There are many types and approaches to version control.
Here we will introduce one implementation: git.

There are a few things we will cover in order to get started with git:

0. How to get git;
1. How to initialize a git repository;
2. How to select files to be committed;
3. How to commit them to the repository;
4. The difference between the repo and working copies;
5. How to delete files from the repository;
6. How to temporarily return to an older version;
7. Where to find more information.

First things first: getting git

- Linux Desktop
  - `sudo dnf/.../apt-get install git`
- MacOS
  - Xcode
  - fink/macports/homebrew
  - git OSX installer
- Windows (MobaXterm)
  - MobaXterm: `apt-get install git`
  - https://gitforwindows.org
- SciNet clusters (Teach, Niagara, . . .)
  - Git is already installed.

Version control: Setup your identity

The first time you use git, it might complain if it can't identify who you are!

Best to set identify your global self to git:

```
$ git config --global user.email "rzon@scinet.utoronto.ca"
$ git config --global user.name "Ramses van Zon"
```

Then git will mark any commits you make with your user name and email.

You can also set other preferences, e.g.

```
$ git config --global init.defaultBranch main
```

This ensures any initial branch will be called 'main' (instead of the default 'master').

Version control: Create a local repository

The first thing to do is initialize a repository for your code.

```
$ mkdir code # if there is no  code yet

$ cd code

$ git init --initial-branch main  # creates a repository for this directory, in the 'main' branch
Initialized empty Git repository in /home/s/scinet/rzon/code/.git/
```

This created a `.git` directory in the current directory.

`.git` ***is*** your local repository (currently empty).

The current directory contains the working copy (currently empty even if there are files in it) .

Note: You cannot see the .git directory with `ls` *unless you give the* `-a` *option, i.e.*

```
$ ls -a
. .. .git
$
```

Version control: adding files to the repository

First you must **add** the files to the hidden *staging area*, then you commit:

```
$ echo "some data" > file1.txt
$ cp file1.txt file2.txt
$ cp file1.txt file3.txt
$ ls
file1.txt file2.txt file3.txt
```

```
$ git add file1.txt file2.txt  # leave out file3.txt for now
```

```
$ git commit -m "First commit for my repository"
[main (root-commit) dd9d139] First commit for my repository
 2 files changed, 2 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
$
```

Note: The working directory is not tracked!

Even if you "git add" a directory, only the content at that time is staged.
You see that here as, `file3.txt` did not make it into the repository.

Version control: Comparing file versions

Let's update some data and see how can we compare it with the already commited files...

```
$ echo "some more data" >> file1.txt
```

```
$ git diff file1.txt
diff --git a/file1.txt b/file1.txt
index 4268632..fdd9353 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1,2 @@
 some data
+some more data
```

We're satisfied and want to add this change:

```
$ git add file1.txt
$ git commit -m "updating data due to ..."
```

Files already in the repo are tracked and we can commit the changes with one command:

```
$ git commit -a -m "updating data due to ..."
```

Always add a descriptive message!(the examples above are bad!)

Version control: Status report

Review what has been done in the repo: git log

```
$ git log
commit b0292f6e3a820856f1d29b5aee2acdc4fd9e73c9 (HEAD -> main)
Author: Ramses van Zon <rzon@scinet.utoronto.ca>
Date:   Thu Jan 27 09:50:01 2022 -0500

    updating data due to ...

commit dd9d13999ac5073089e6ea4282b0c78854256bc1
Author: Ramses van Zon <rzon@scinet.utoronto.ca>
Date:   Thu Jan 27 09:49:02 2022 -0500

    First commit for my repository
```

Version control: Detailed status report

Commits are in reverse chronological order.

Each commit has a hexadecimal hash.
b0292f6e3a820856f1d29b5aee2acdc4fd9e73
dd9d13999ac5073089e6ea4282b0c7885425

Detailed review what has been done: git log –stat

```
$ git log --stat
commit b0292f6e3a820856f1d29b5aee2acdc4fd9e73c9 (HEAD -> main)
Author: Ramses van Zon <rzon@scinet.utoronto.ca>
Date:   Thu Jan 27 09:50:01 2022 -0500

    updating data due to ...

 file1.txt | 1 +
 1 file changed, 1 insertion(+)

commit dd9d13999ac5073089e6ea4282b0c78854256bc1
Author: Ramses van Zon <rzon@scinet.utoronto.ca>
Date:   Thu Jan 27 09:49:02 2022 -0500

    First commit for my repository

 file1.txt  | 1 +
 file2.txt | 1 +
 2 files changed, 2 insertions(+)
```

Version control: Working and staging area status

To check the status of files in the working and staging area, use

```
$ git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed
    file3.txt

nothing added to commit but untracked files present (use "git a
```

Look what happens if we add a file

```
$ git add file3.txt
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   file3.txt
```

Inspecting a specific previous version

To look at a previous commit temporarily, use the commit hash and do

```
$ git checkout dd9d13999ac5073089e6ea4282b0c78854256bc1 -b tmpbr
```

The `-b` option creates a temporary branch called `tmpbr`, leaving the main branch intact.

Without the `-b tmpbr` option, we see the same files but we'd lose track of the real HEAD. Unless you jotted down the commit hash of commit you were at, it's hard to get back.

But because we made this a branch, there is an easy way back to the most recent version:

```
$ git checkout main
$ git branch --delete tmpbr
```

Rollback

If you have edited files but something broke and you have not yet committed,

you can bring the working copy back to the most recent commit:

```
$ git reset --hard
```

Rolling back to a specific previous version, you can be done with a hard reset:

```
$ git reset dd9d13999ac5073089e6ea4282b0c78854256bc1 --hard
```

This erases the part of the history after `dd9d13999ac5073089e6ea4282b0c78854256bc1`!

To go to the previous version:

```
$ git reset HEAD^ --hard
```

`HEAD` is the last commit in the history, `HEAD^` the one before that, `HEAD^^` the one before that etc.

Without the `--hard` option, only the repo in `.git` is updated, but the files in the working directory would not have been restored.

Note: Actually all the commits are still there in the repo, but without references to it, so they are effectively gone (and eventually deleted).

Reverting changes – Reset, Checkout, and Revert

```
git reset <Commit>           # Throw away uncommited changes (consider '--hard')
git restore --staged <file> # Unstage a file i.e. undo an 'add'
git checkout <Commit>        # Inspect old commit (consider a '-b tmpbr' option)
git checkout <Branch>        # Switch between branches
git checkout <File>          # Discard changes in the working directory
git revert <Commit>          # Create a new commit that undoes the given commit
```

By the way: nobody pretends git command are intuitive.

Version control: removing repository files

Let's look at what we've done so far.

```
# git log
commit b0292f6e3a820856f1d29b5aee2acdc4fd9e73c9 (HEAD -> main)
Author: Ramses van Zon <rzon@scinet.utoronto.ca>
Date:   Thu Jan 27 09:50:01 2022 -0500
    updating data due to ...

commit dd9d13999ac5073089e6ea4282b0c78854256bc1
Author: Ramses van Zon <rzon@scinet.utoronto.ca>
Date:   Thu Jan 27 09:49:02 2022 -0500
    First commit for my repository
$
```

If you want to delete a file in the working copy and the repo, use `git rm`:

```
$ git rm file2.txt  # let git do it!
$ git commit -m "Remove file2.txt"
[main f1af560] removed file2.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 file2.txt
$
```

Version control: Git branches

- Show current branch

```
$ git branch
```

- Show all branches

```
git branch -a
```

- Creates a branch

```
git branch MYNEWBRANCH
```

- Switch to the branch

```
git checkout branchname
```

- Shows all remote branches

```
git branch -r
```

Remote repositories

Remote repositories

- Git is a distributed version control system.
- You can clone a repo anywhere to copy it elsewhere.
- Each clone is a full-fledged repo with full history.
- You can push and pull the state of one repo to another.
- You do not have to have one centralized, authoritive repo, but often, that is still convenient.
- Clones can live on remote computers or in the cloud (e.g. github, gitlab)
- Git can interact with remote repos using `ssh` (as well in other ways).
- Remote repos often don't need a working directory, they can be bare .git repos.

Remote repositories: Clone, pull, push

1. Setup a remote repo on a cluster

```
local> ssh USERNAME@teach.scinet.utoronto.ca
teach> mkdir repo
teach> cd repo
teach> git init --initial-branch main
teach> git config receive.denyCurrentBranch updateInstead
teach> echo "hello" > world.txt
teach> git add world.txt
teach> git commit -m 'hello world'
teach> exit
```

2. Clone it on your local computer:

```
local> git clone USERNAME@teach.scinet.utoronto.ca:repo
local> cd repo
local> echo "more" >> file1.txt
local> echo "most" >> file2.txt
local> git add file1.txt file2.txt
local> git commit -m "Added files"
```

3. Update the repo in teach:

```
local> git push -u origin main
```

Check:

```
local> ssh USR@teach.scinet.utoronto.ca
teach> cd repo
teach> git reset --hard
teach> ls
world.txt file1.txt file2.txt
```

4. Make changes on teach

```
teach> echo "even more data" >> file1.txt
teach> git add file1.txt
teach> git commit -m 'More data added'
```

5. Update repo locally:

```
local> git pull
```

Public remote repositories

You have probably heard of web-based options for git as well:

- GitHub: https://github.com
- GitLab: https://gitlab.com
- Bitbucket: https://bitbucket.org

These services can host your repos as a remote repo, and make them publicly available (or not).

They typically allow access through ssh by setting up ssh keys.

If your repo is public, consider using a `main` and a `development` branch.

Git != GitHub

While these services use git, they add functionality and workflows.

For instance, in addition to branches, they support

- Forks: a personal copy of a repo that you can modify without affective the original.
- Pull requests: a proposal to merge your changes into the original repo.

Version control: a few tips

- Use it, it will save you trouble.
- Commit often.
- Include sensible commit messages.
- It's easy to forget to add files, check "git status" or clone to another directory as a check.
- But do not commit derivative stuff (e.g. log files, executables, compiled modules, ...)
- Can be used for several different kind of projects: code development, collaborations, papers, ...
- There are other VC systems: hg, svn, cvs, ...

Tutorials

- https://www.vogella.com/tutorials/Git/article.html
- https://www.atlassian.com/git/tutorials