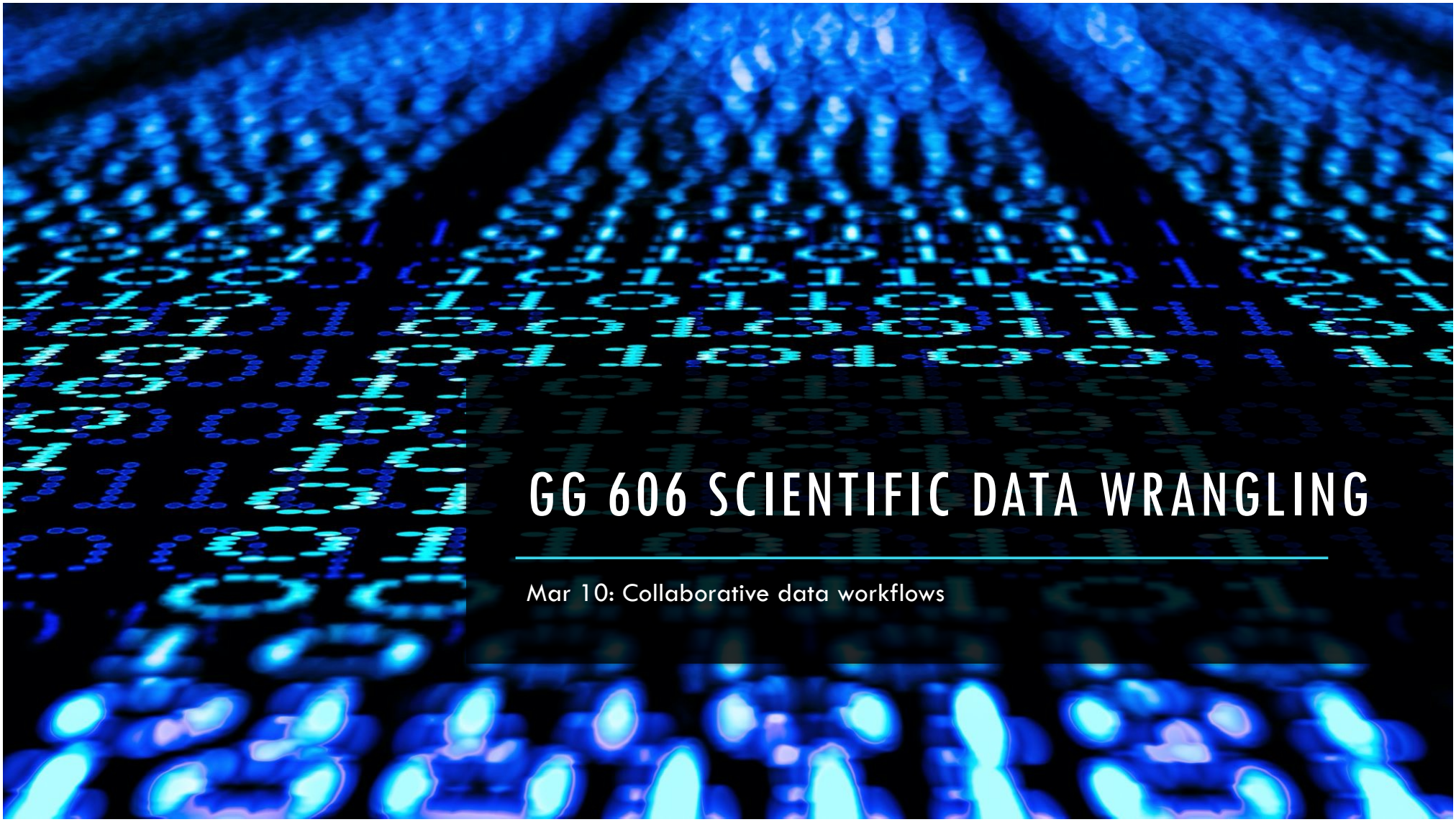


GG 606 SCIENTIFIC DATA WRANGLING

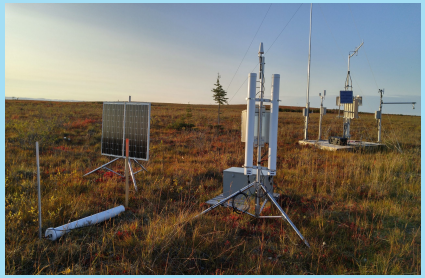
Mar 10: Collaborative data workflows

THE LONELY ANALYST SCIENTIFIC DATA WORKFLOW

Your Collaborators

Government websites/APIs

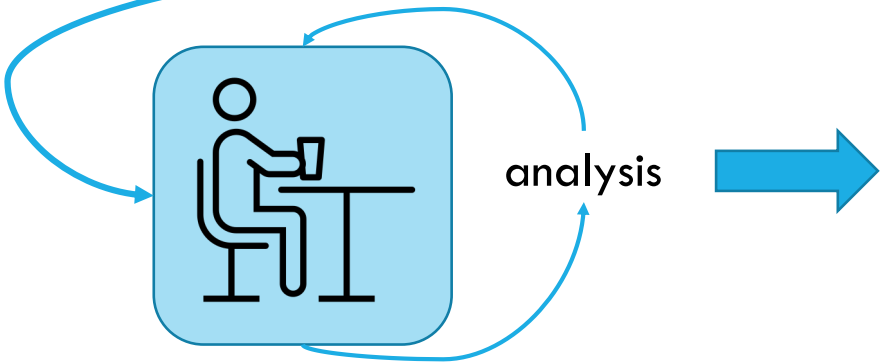
Your sensors/instruments



Your supervisor

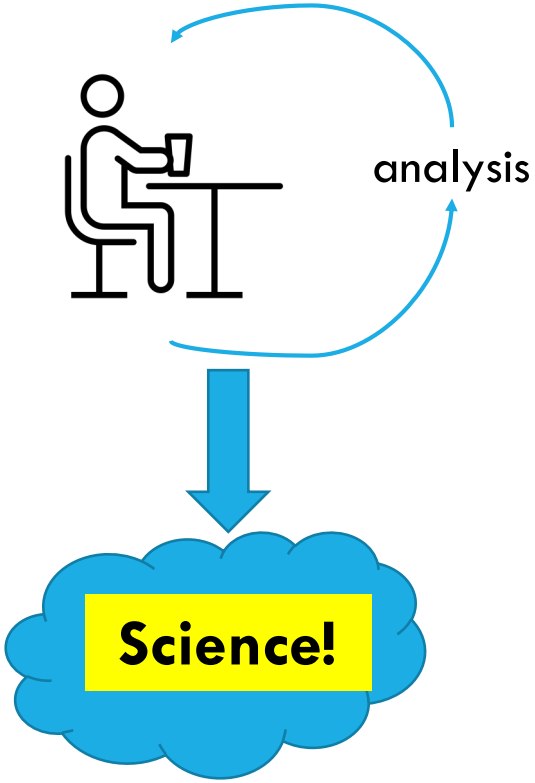
Your colleagues

You



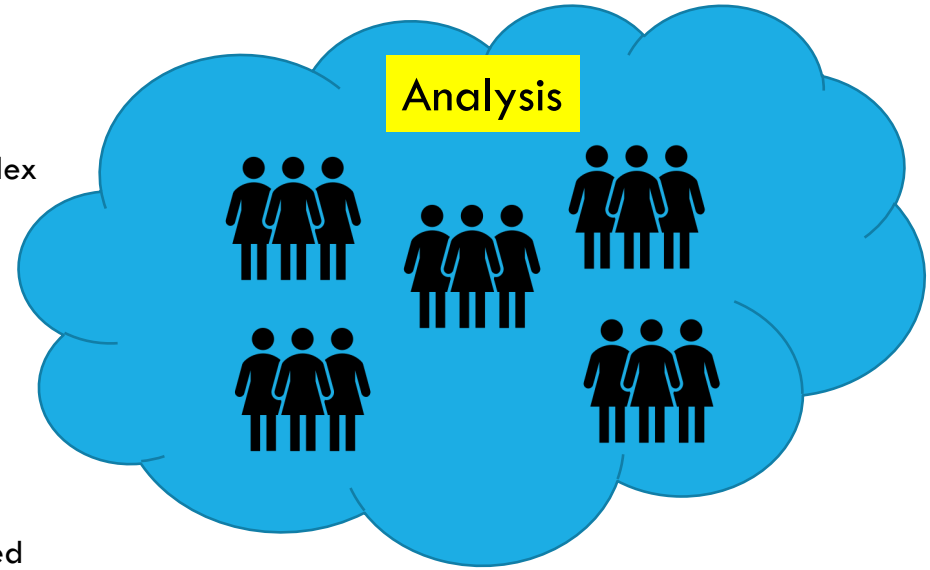
PROBLEMS WITH THIS MODEL

- It is different from many types of real-world data-analytic work
  - almost all work in industry
  - large research projects
  - open source and community projects
- Not transparent: Only you know in detail what is happening in your analysis
  - OK for graduate research
  - Not great for much else
- **The real world is collaboratory**
  - this is a good thing



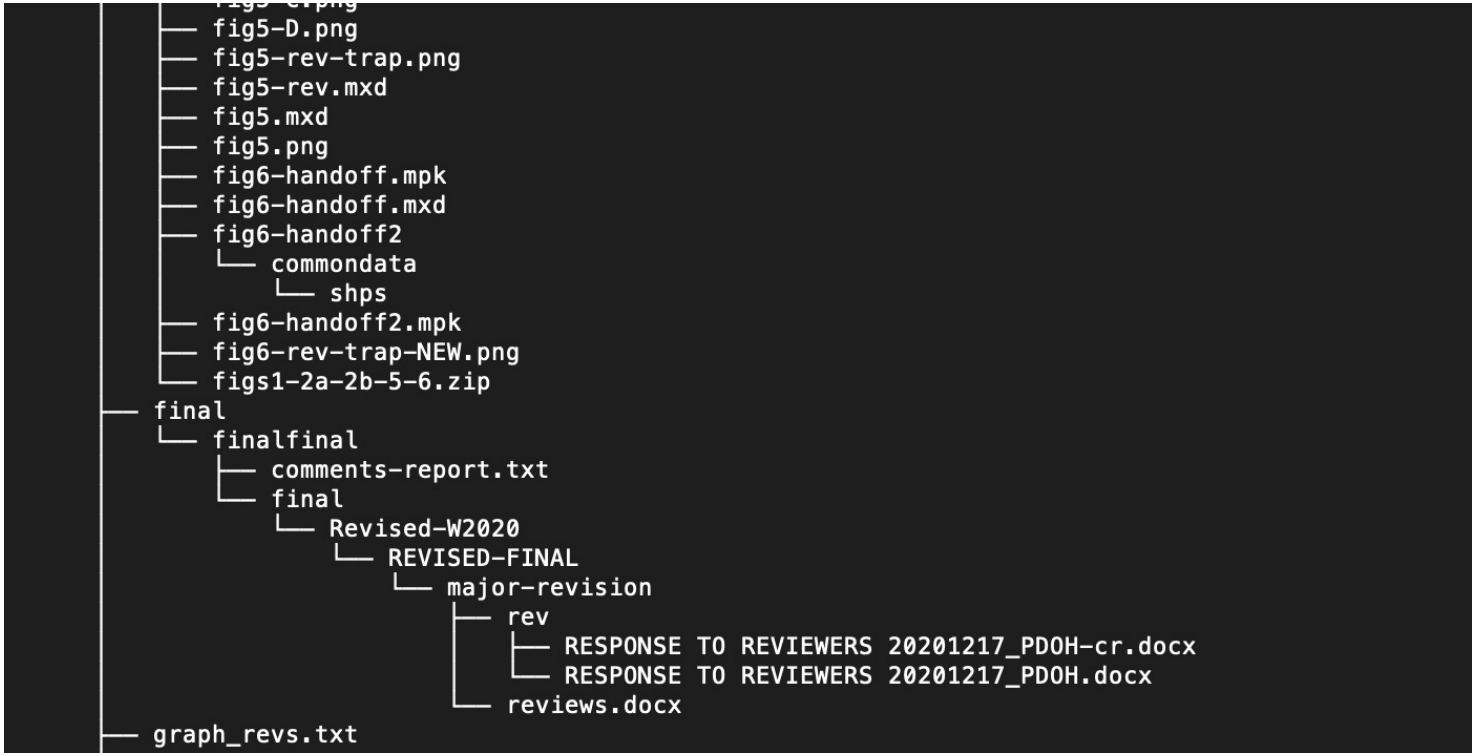
COLLABORATIVE SCIENCE

- Participatory
  - inputs from multiple people adding substantively to complex analysis and modelling
- Transparent
  - supports reproducible science workflows by being code-based and with supporting documentation
  - can be adopted and maintained and contributed to when you go on vacation
- Change management
  - changes are tracked, versions are stored and can be rolled back
- Redundant
  - backed up and tolerant to system outages, network failures, things generally breaking
    - because they do



HOW DO WE DO COLLABORATIVE DATA WORK

- Using copy, paste, file renaming and directories for version control has limitations



COLLABORATION TOOLS

- There are tools available for collaborative work of different kinds

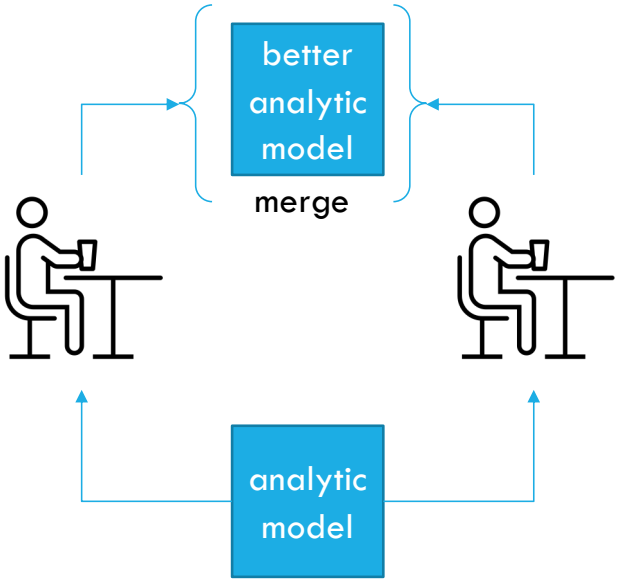
following materials adapted from D. Matuszek, A. West and K. Pu

VERSION CONTROL SYSTEMS

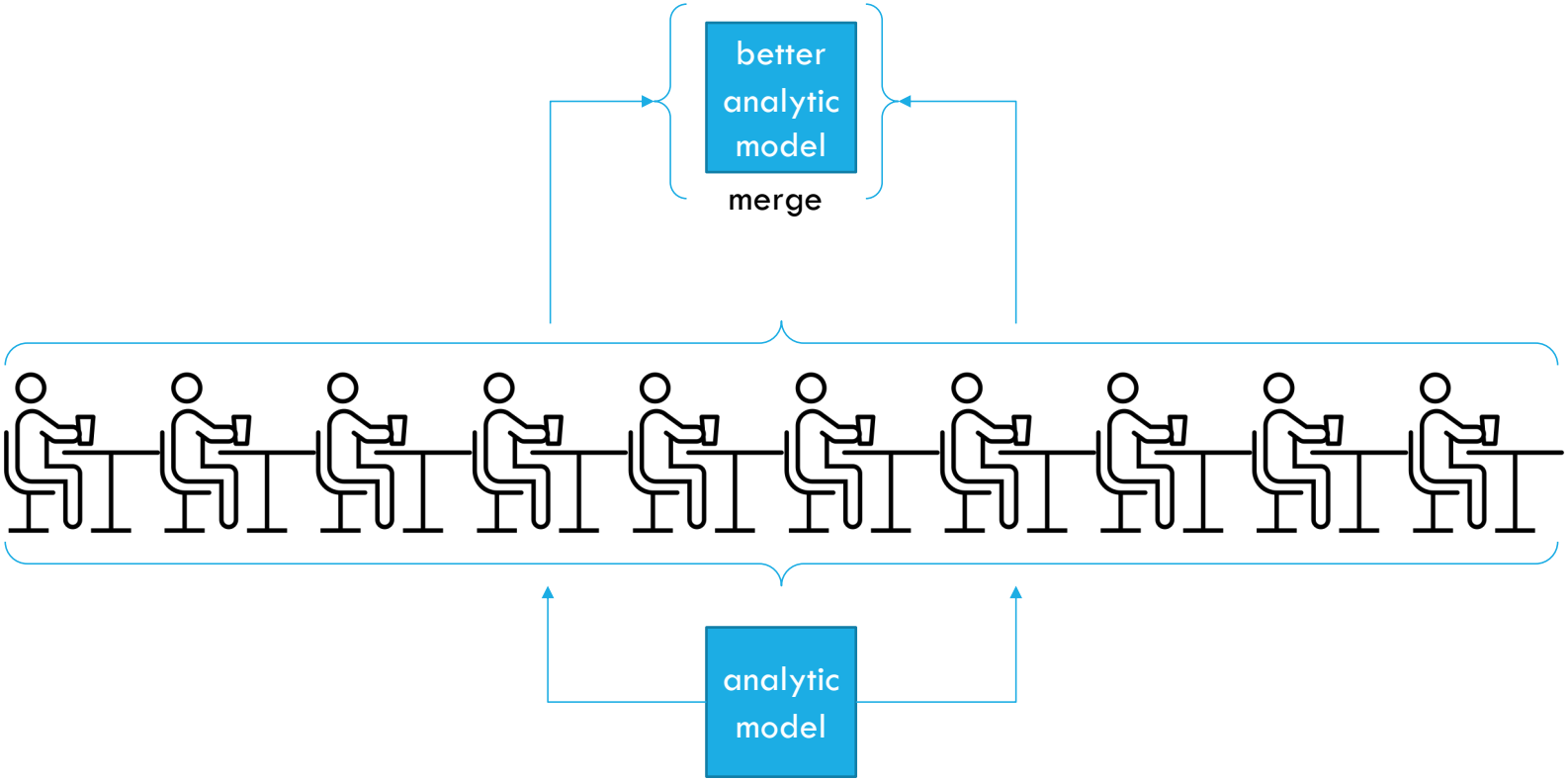
- Version control (or revision control, or source control) is all about managing multiple versions of documents, programs, web sites, etc.
  - Almost all "real" projects use some kind of version control
  - open source software, collaborative scientific models, etc
  - Essential for team projects, but also very useful for individual projects
- Some well-known version control systems are CVS, Subversion, Mercurial, and Git
  - CVS and Subversion use a "central" repository; users "check out" files, work on them, and "check them in"
  - Mercurial and Git treat all repositories as equal

WHY VERSION CONTROL?

- For working by yourself:
  - Gives you a "time machine" for going back to earlier versions
  - Gives you great support for different versions (standalone, web app, etc.) of the same basic project
- For working with others:
  - Greatly simplifies concurrent work
  - merging changes
- Without version control, collaboration is only feasible for very small teams



WHY VERSION CONTROL?



DOWNLOAD AND INSTALL GIT

- There are many available online materials for getting started with git
- Here's the standard one: http://git-scm.com/downloads
- Here is another https://rogerdudler.github.io/git-guide/

- Note: Git is primarily a command-line tool

INTRODUCE YOURSELF TO GIT

- Enter these lines (with appropriate changes):
  - `git config --global user.name "Colin Robertson"`
  - `git config --global user.email crobertson@wlu.ca`
- You only need to do this once

- If you want to use a different name/email address for a particular project, you can change it for just that project
  - `cd` to the project directory
  - Use the above commands, but leave out the `--global`

CREATE AND FILL A REPOSITORY

1. `cd` to the project directory you want to use
2. Type in `git init`
   - This creates the repository (a directory named `.git`)
   - You seldom (if ever) need to look inside this directory
3. Type in `git add .`
   - The period at the end is part of this command!
     - Period means "this directory"
   - This adds all your current files to the repository
4. Type in `git commit -m "Initial commit"`
   - You can use a different commit message, if you like

CLONE A REPOSITORY FROM ELSEWHERE

- `git clone` *`URL`*
- `git clone` *`URL mypath`*
  - These make an exact copy of the repository at the given URL
- `git clone git://github.com/`*`rest_of_path/file.git`*
  - Github is the most popular (free) public repository
- All repositories are equal
  - But you can treat some particular repository (such as one on Github) as the "master" directory
- Typically, each team member works in his/her own repository, and "merges" with other repositories as appropriate

THE REPOSITORY

- Your top-level **working directory** contains everything about your project
  - The working directory probably contains many subdirectories—source code, binaries, documentation, data files, etc.
  - One of these subdirectories, named .git, is your repository
- At any time, you can take a "snapshot" of everything (or selected things) in your project directory, and put it in your repository
  - This "snapshot" is called a commit object
  - The commit object contains (1) a set of files, (2) references to the "parents" of the commit object, and (3) a unique "SHA1" name
  - Commit objects do *not* require huge amounts of memory
- You can work as much as you like in your working directory, but the repository isn't updated until you `commit` something

INIT AND THE .GIT REPOSITORY

- When you said `git init` in your project directory, or when you cloned an existing project, you created a repository
  - The repository is a subdirectory named .git containing various files
  - The dot indicates a "hidden" directory
  - You do *not* work directly with the contents of that directory; various git commands do that for you
  - You *do* need a basic understanding of what is in the repository

MAKING COMMITS

- You do your work in your project directory, as usual
- If you create new files and/or folders, they are *not tracked* by Git unless you ask it to do so
  - `git add newFile1 newFolder1 newFolder2 newFile2`
- Committing makes a "snapshot" of everything being tracked into your repository
  - A message telling what you have done is required
  - `git commit -m "fixed typo in function description"`
  - `git commit`
    - This version opens an editor for you the enter the message
    - To finish, save and quit the editor
- Format of the commit message
  - One line containing the complete summary
  - If more than one line, the second line must be blank

COMMITS AND GRAPHS

- A commit is when you tell git that a change (or addition) you have made is ready to be included in the project
- When you commit your change to git, it creates a commit object
  - A commit object represents the complete state of the project, including all the files in the project
  - The *very first* commit object has no "parents"
  - Usually, you take some commit object, make some changes, and create a new commit object; the original commit object is the parent of the new commit object
    - Hence, most commit objects have a single parent
  - You can also merge two commit objects to form a new one
    - The new commit object has two parents
- Hence, commit objects form a **directed graph**
  - Git is all about using and manipulating this graph

WORKING WITH YOUR OWN REPOSITORY

- A head is a reference to a commit object
- The "current head" is called HEAD (all caps)
- Usually, you will take HEAD (the current commit object), make some changes to it, and commit the changes, creating a new current commit object
- You can also take any previous commit object, make changes to it, and commit those changes
  - This creates a branch in the graph of commit objects
- You can merge any previous commit objects
  - This joins branches in the commit graph

COMMIT MESSAGES

- In git, “Commits are cheap.” Do them often.
- When you commit, you must provide a one-line message stating what you have done
  - Terrible message: “Fixed a bunch of things”
  - Better message: “Corrected the calculation of median scores”
- Commit messages can be very helpful, to yourself as well as to your team members
- You can’t say much in one line, so commit often

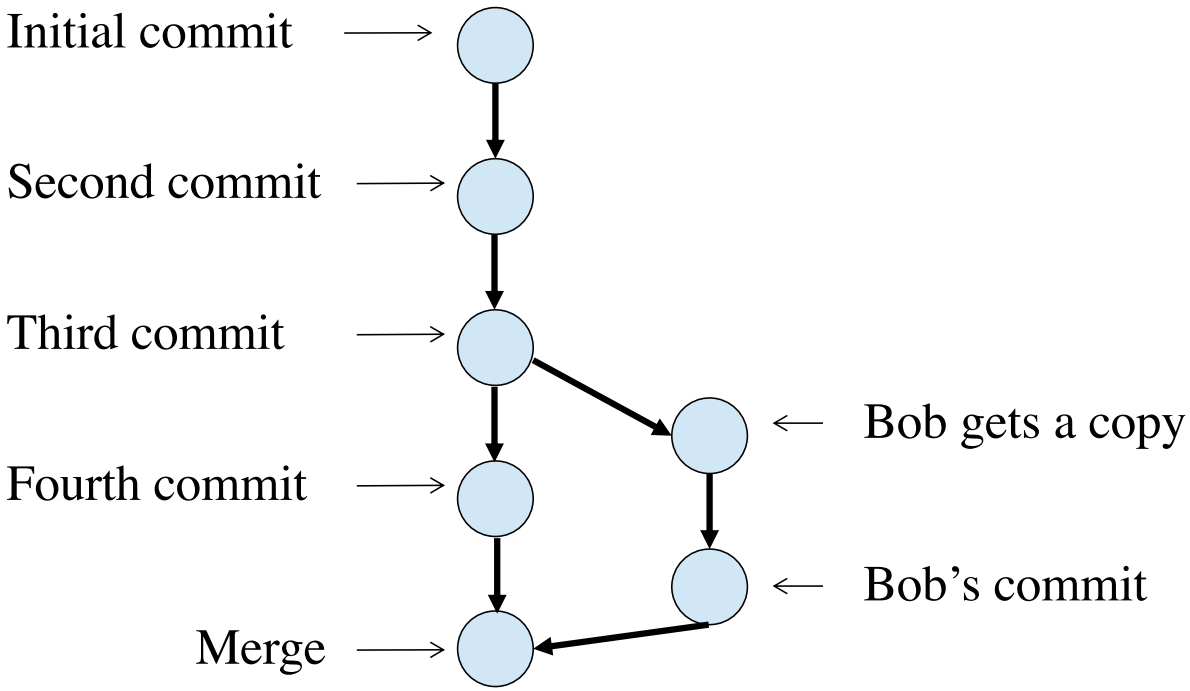
WORKING WITH OTHERS

- All repositories are equal, but it is convenient to have one central repository in the cloud
- Here's what you normally do:
  - Download the current HEAD from the central repository
  - Make your changes
  - Commit your changes to your local repository
  - Check to make sure someone else on your team hasn't updated the central repository since you got it
  - Upload your changes to the central repository
- If the central repository *has* changed since you got it:
  - It is *your* responsibility to **merge your two versions**
    - This is a strong incentive to commit and upload often!
  - Git can do this for you, if there aren't incompatible changes

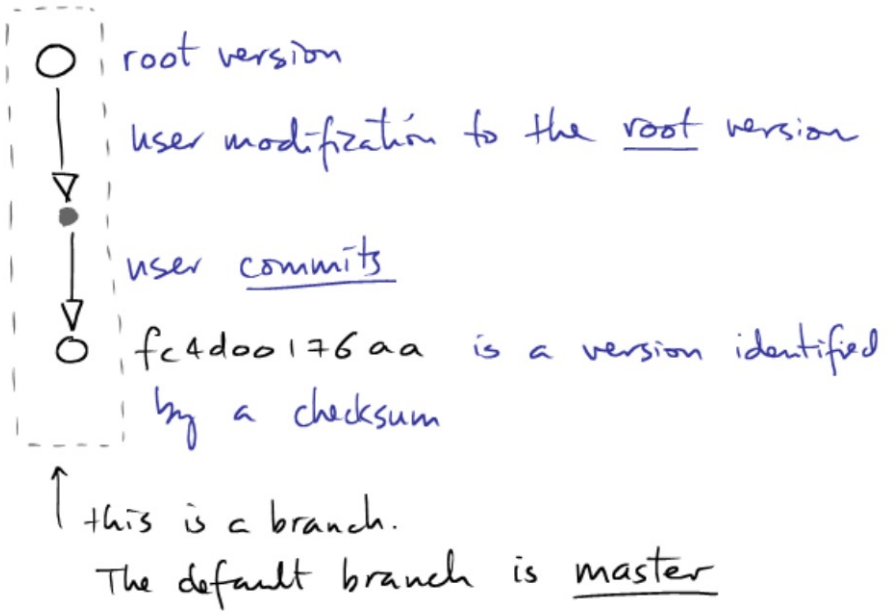
TYPICAL WORKFLOW

- `git pull` *`remote_repository`*
  - Get changes from a remote repository and merge them into your own repository
- `git status`
  - See what Git thinks is going on
  - Use this frequently!
- Work on your files (remember to add any new ones)
- `git commit -m` *`"What I did"`*
- `git push`

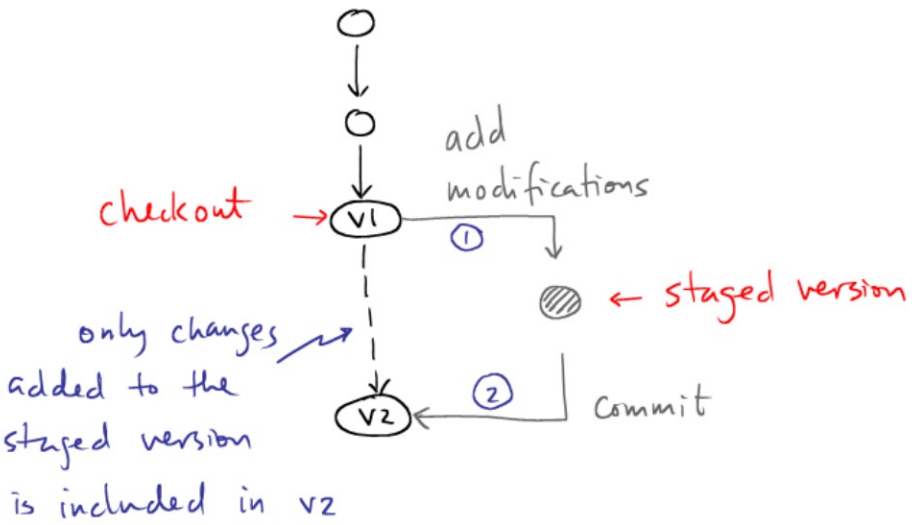
MULTIPLE VERSIONS



Workflow: version control




```
##
# create an empty repo
##
git init          # a local repo to work with
git init --bare   # a _bare_ repo that can be shared
```



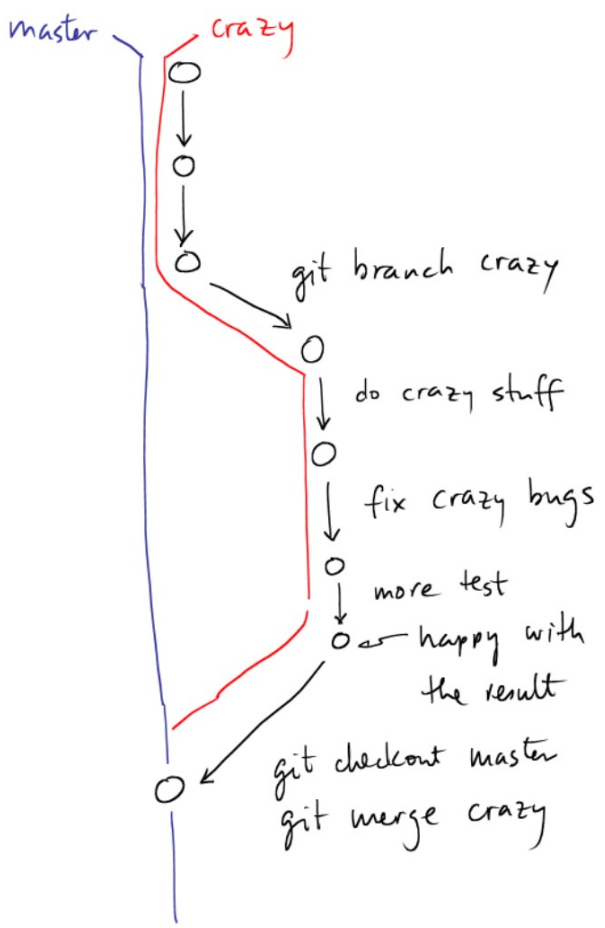
Git ignores *all* changes to files on disk unless you explicitly tell git to include some change into the repository.

```
vi important_algorithm.cc

##
# Add imporant_algorithm.cc to the staging version
##
git add important_algorithm.cc

##
# Commit the staged files to create a new version.
##
git commit -m 'important changes to important_algo
rithm.cc'
```

Workflow: trying things out -- branching

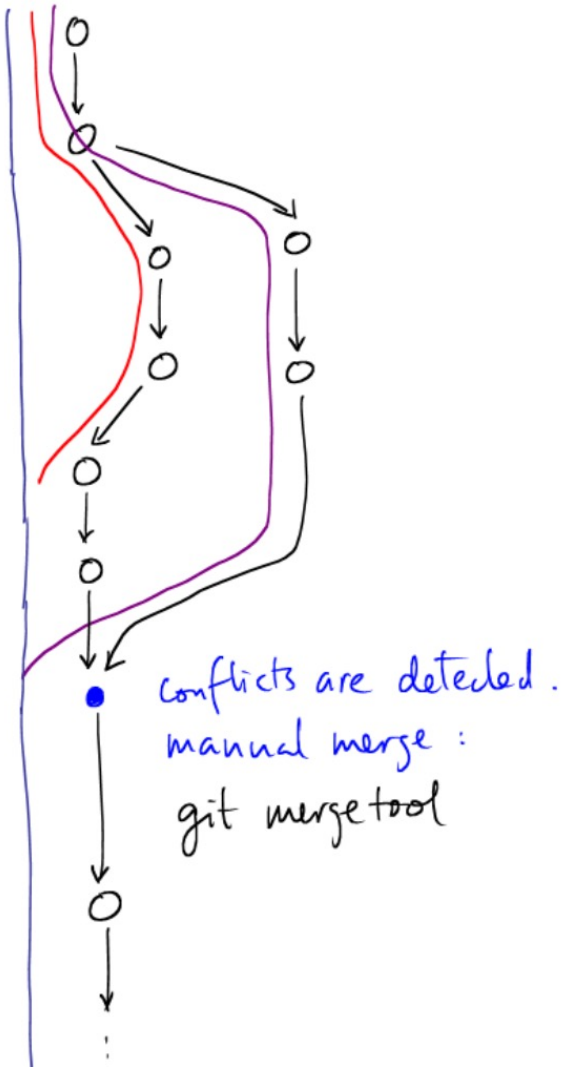



```
$ git branch
master
$ git branch crazy
You are on crazy branch

##
# We are on the crazy branch
##
$ git branch
master
* crazy
$ vi important_algorithm.cc
$ git commit -m 'made some crazy changes, not sure'
$ ...
$ git commit -m 'yes, we are really happy with the changes'

##
# Switch back to the master
# (all changes disappears from working directory
##
git checkout master

```



If multiple people are working on the same lines of text, conflicts will occurr, and the merge tool probably cannot safely resolve them during `git merge`.

Git offers a lot of nice tools to help out:

- `git diff master crazy`
  This helps to see the updates made in `crazy` before merge is done.
- If `git merge crazy` fails, the conflicts are marked *in the files*, waiting for manual intervention. If you hate resolving the conflicts, just undo the merge.
  - check out the older version of conflicting files, *or*
  - `git merge --abort`
- Sometimes, you just have to do the merge yourself. Use a mergetool: `git mergetool` launches the mergetool you configured automatically.

KEEPING IT SIMPLE

- If you:
  - Make sure you are current with the central repository (use `git pull`)
  - Make some improvements to your code
  - Commit + push to the repo
- Work on isolated small parts of the codebease so you don't have to resolve conflicts or work with multiple branches
  - All the complexity in git comes from dealing with these
- Therefore:
  - Make sure you are up-to-date before starting to work
  - Commit and update the central repository frequently
- If you need help: https://help.github.com/

REFERENCES

- http://www.cs.toronto.edu/~kenpu/articles/cs/git-intro.html
- Tutorial to try:

  https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners

GG 606 SCIENTIFIC DATA WRANGLING

Mar 10: Collaborative data workflows