CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 4

Thursday, September 16th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Introduction to version control systems
- Centralized and distributed approaches
- Bare essentials of `git`
- `git rebase` demo (rebasing is important to restore tidiness after you've been messy) [today]

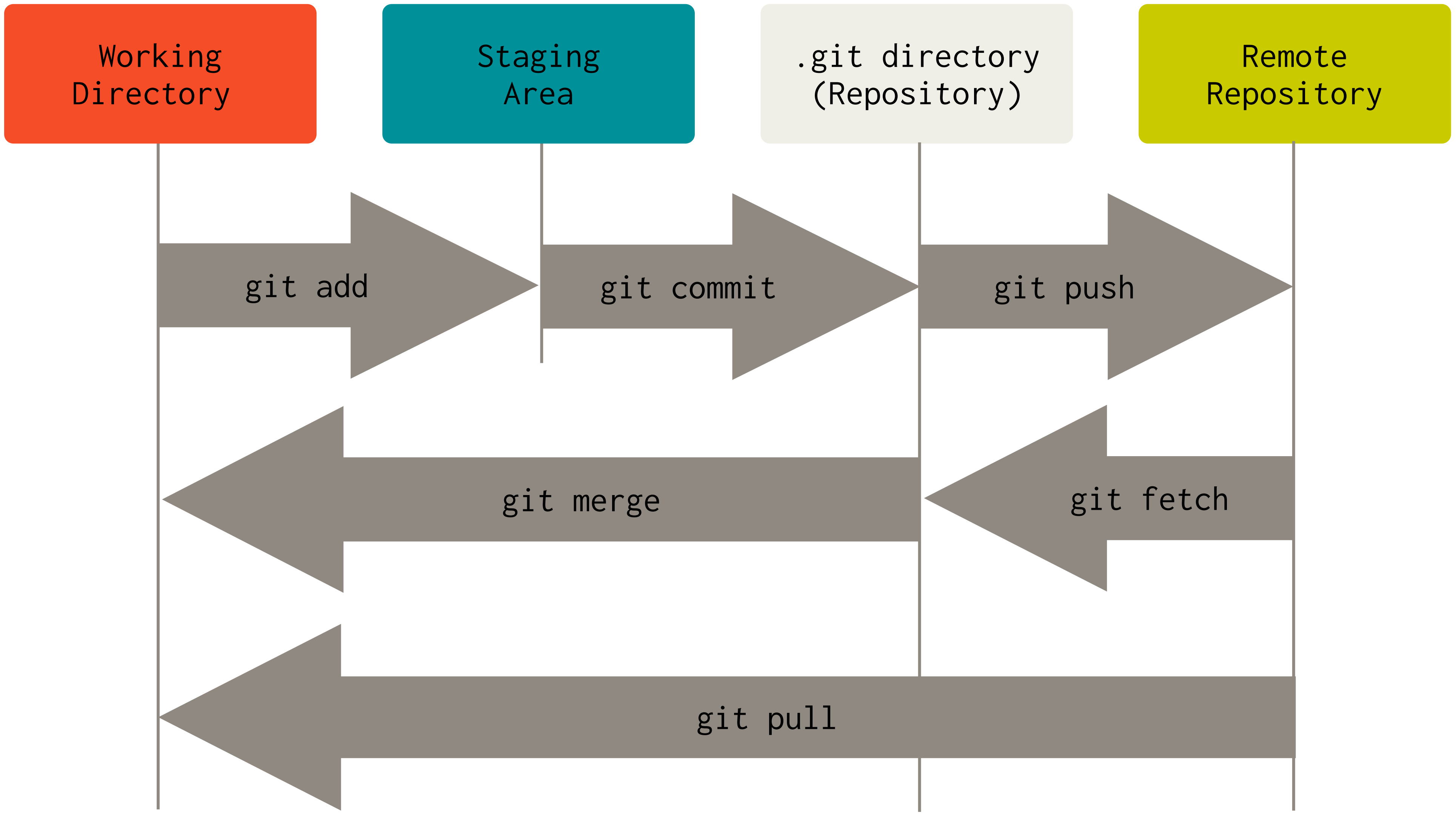
OUTLINE

- More `git` basics
- Remote repositories
- Branching in `git`

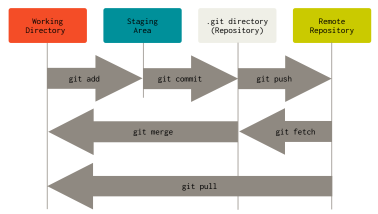
Content and some figures are based on the free Pro Git book written by Scott Chacon and Ben Straub.

MORE `git` BASICS

BASIC `git` COMMANDS YOU MUST INTERNALIZE



BASIC `git` COMMANDS YOU MUST INTERNALIZE



can use: git commit -a : all (will not add untracked files)

- `git add` : add new or modified files to the index (staging area in the `.git/index` file)

Remark: you could use "`git add .`" to add *any* new or modified files in one go. This is ***bad practice*** because it may add files to the index that you did not intend to. Your colleagues will not be happy about this. Only lazy people do this.

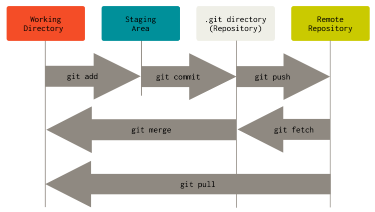
- `git commit` : commit the staged changes to the repo

Remark: It is ***good practice*** to create small, well-arranged commits. You can always *rebase* if you think two (or more) small commits belong to one commit.

- `git push` : push commits to the upstream repository

Remark: The upstream repository never has a working directory checked out. It only consists of the contents inside the `.git` directory. It can be on a remote location or locally (e.g. for backup purposes). See the `--bare` option of `git help init`.

BASIC `git` COMMANDS YOU MUST INTERNALIZE



- `git fetch`: fetch new commits from the upstream repository (e.g. from your collaborators)

  Remark: This will only update your local `.git` repository, not your working directory.

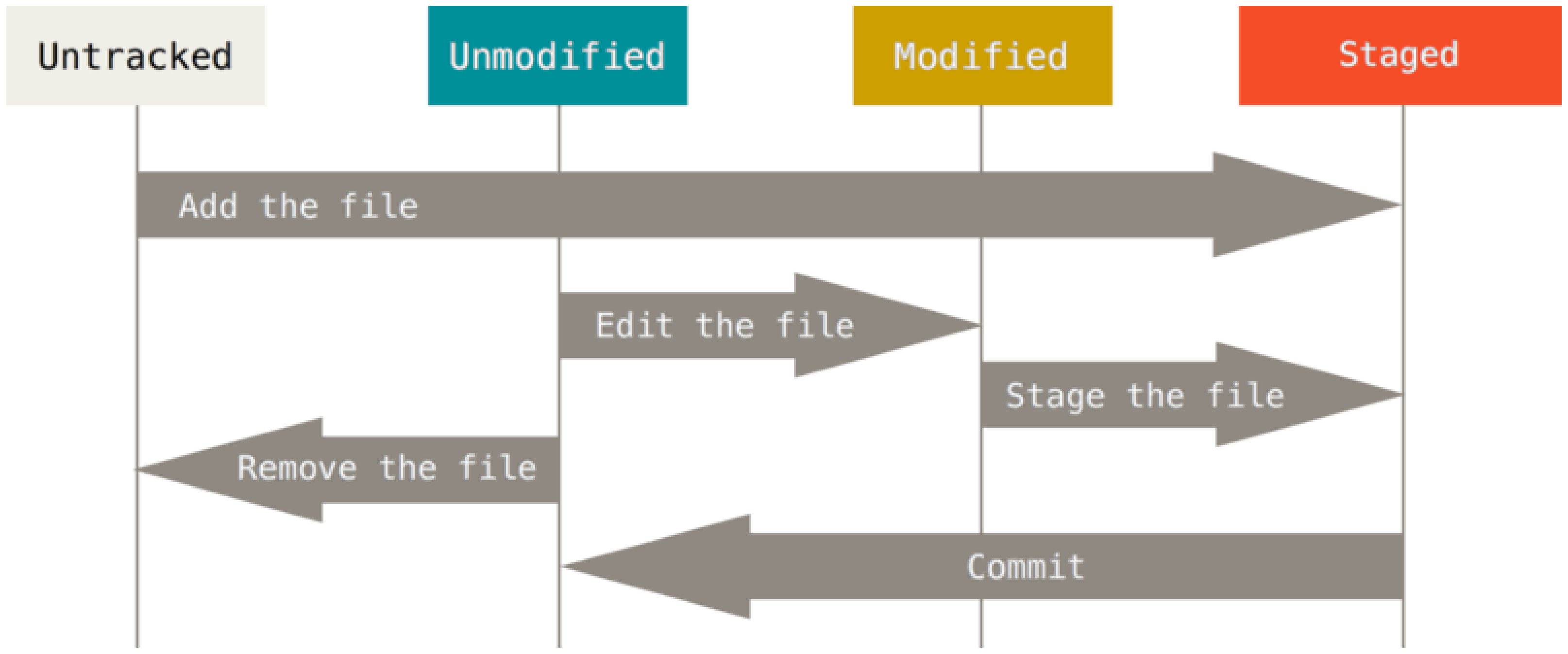
- `git merge`: update your working directory by merging new commits from your local repository (joining histories)

  Remark: By default this merges the branch you are currently on. You can merge any other branches you like, we will see this in a few slides from now.

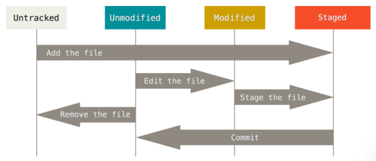
- `git pull`: fetch commits from the upstream repository and merge them with the current working directory

  Remark: This saves you some time as most often you want this behavior, rather than executing `git fetch` followed by `git merge`.

UNDERSTAND THE STATUS OF YOUR FILES



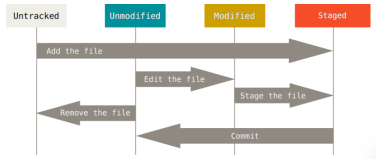
UNDERSTAND THE STATUS OF YOUR FILES



From ***untracked*** to ***staged***:

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file

nothing added to commit but untracked files present (use "git add" to
$ git add file
$ git status
On branch master

No commits yet

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

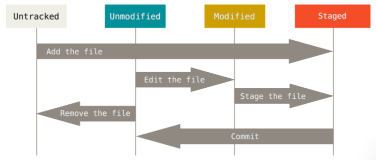
UNDERSTAND THE STATUS OF YOUR FILES



From *staged* to *unmodified*: record the snapshot

```
$ git commit -m "Added untracked file"
[master (root-commit) 68581f7] Added untracked file
 1 file changed, 1 insertion(+)
 create mode 100644 file
$ git status
On branch master
nothing to commit, working tree clean
```

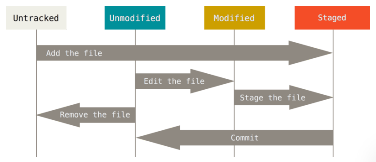
UNDERSTAND THE STATUS OF YOUR FILES



From ***unmodified*** to ***modified***: edit the tracked file

```
$ echo 'Adding a new line of text' >> file
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory
        modified:   file

no changes added to commit (use "git add" and/or "git commit -a")
```

UNDERSTAND THE STATUS OF YOUR FILES

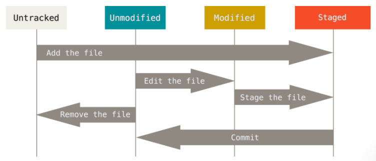


From ***modified*** to ***staged***: add the file to the index

```
$ git add file
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file
```

- We can now run `git commit` again to go from staged to unmodified (by recording a new snapshot)
- Instead of running `git add file` and then commit, we can combine these steps by
  `git commit -am <commit message>`
- "`git commit -a`" ***is not*** the same as "`git add .`" followed by `git commit` (the latter adds *untracked* files too!)

UNDERSTAND THE STATUS OF YOUR FILES

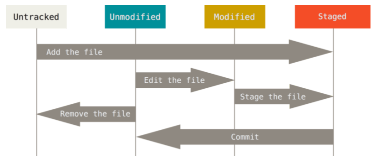


If you modify a ***staged*** file again:

```
$ echo "Modify a staged file" >> file
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working director
        modified:   file
$ git add file
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file
```

- New modifications are separate from the ones you have staged earlier
- If they belong in the same commit, then you need to run `git add file` again to add your new changes to the already staged changes!

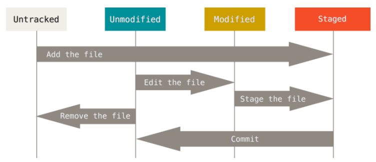
UNDERSTAND THE STATUS OF YOUR FILES



You can remove tracked files from the repository. Removing files has two faces in `git`:

1. Remove files from the `.git` repository, keep them in your file system
2. Remove files from both, `.git` repository ***and*** your file system.

UNDERSTAND THE STATUS OF YOUR FILES



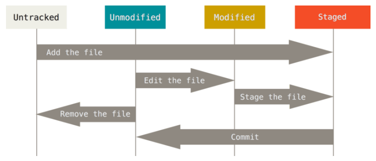
1. Remove files from the `.git` repository, keep them in your file system

only know the files that are tracked

```
$ git ls-files  # list files that are known to git
file
$ git rm --cached file
rm 'file'
$ git ls-files  # no output = no files tracked
$ ls
file  # the file is still with us, but not under VCS anymore
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file
```

- We still need to commit the changes, even when we delete files from the repository. ***Remember:*** `git` thinks in terms of file systems, you must record a snapshot when you remove files too.

UNDERSTAND THE STATUS OF YOUR FILES

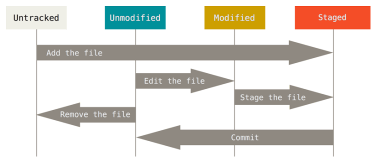


1. Remove files from the `.git` repository, keep them in your file system
2. Remove files from both, `.git` repository ***and*** your file system.

```
$ git rm file
rm 'file'
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file
```

- This time the file is gone (`git` does not report untracked files in the current directory)
- But fear not, we can still restore any removed files from the `git` file system snapshot using "`git restore`"
- There were couple of times where `git` really saved me from a lot of trouble thanks to restoring...

UNDERSTAND THE STATUS OF YOUR FILES



- Note that `git` uses the same command name to remove files as on the Linux command line itself: `rm`
- The same applies when you want to rename files. In that case you would use the `mv` command (move) in the Linux command line as well as for `git`: `git mv`

```
$ git mv file new_filename
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    file -> new_filename
```

- To move a file means to change the file system. `git` wants you to commit a snapshot for this action, as usual.

KNOW YOUR HISTORY

- Every commit you make in `git` is recorded in the ***history***.
- The history contains a huge amount of information and obviously is important when you need to comprehend changes that were not necessarily committed by you.
- For that reason, every commit must be documented accordingly.
- *It is not easy to write good commit messages!*

KNOW YOUR HISTORY

Good practices for commit messages:

- The anatomy of a commit message is:

```
Message subject: One single line, (should be) not more than 72 characters

Longer message body: The body provides more details if the commit
contains a complex change.  It can consist of multiple paragraphs,
formatted at 72 characters per line maximum (some projects are very
strict on these format requirements because the commit will go into the
history of the project and it should maintain a consistent format).
These format requirements are usually implemented (or can be configured
easily) in editors like vim or emacs when you write git commit messages.
You may omit the message body if your commit is small and the subject is
descriptive enough.

(The subject line above is actually 74 characters long...)
```

- *Write short and concise message subjects!*

```
$ git log --oneline
9cb047b (HEAD -> master) Add license information in README.md    <- OK
b51859a (origin/master) Add tests                          <- Not very descriptive!
2c1f77c Minor                                              <- This is bad!
5dfb982 Bug fix                                            <- Also bad!
```

KNOW YOUR HISTORY

- You display the history with "`git log`"
- The anatomy of a history entry looks like this:

```
commit 72e96d44caf034fdad447eb40ff9cf001075bd0f
Author: Fabian Wermelinger <author@domain.net>
Date:   Mon Jun 21 18:38:39 2021 +0200

    Add src_field_ and dst_field_ for pointwise kernel inputs

    Memory layout in fields is more favorable for pointwise operations than
    pitched layout in labs.  This allows to test kernels that take a field
    (without ghost cells) as input source.
```

  - Commit identifier (SHA-1 hash)
  - Commit author/committer and date
  - Commit subject
  - Followed by commit body

KNOW YOUR HISTORY

Searching the history: `git log`

- Often you need to search the history for specific keywords, commits or the commit author
- You can specify the `--grep=<pattern>` option to search for a regex pattern. This searches *log messages* (subject or body)
- You can specify the `--author=<pattern>` option to search for a particular author/committer using a regex pattern. This only searches author information but not commit messages.
- If you use the `--grep` option multiple times, any pattern may match. If you want that *all patterns must match* pass the `--all-match` option.

KNOW YOUR HISTORY

Formatting the output: `git log`

- You can change the format of how `git` displays the history log
- Use `git log --pretty=oneline` to display the commits in compact form (this option also exists as `--oneline` because it is used often). If you want a lot of information, you can use `--pretty=fuller` instead. See `git help log` for docs.
- Your `git` installation also ships with a graphical tool that you may use to explore the history. You can use the `gitk` tool or `git gui`

IGNORE DATA THAT YOU DON'T WANT TO TRACK

- Often you will have certain files in your project that you don't want under VCS
- In `git` you can use one or many `.gitignore` files for this purpose
- Recall that files starting with a "`.`" are *hidden* files

IGNORE DATA THAT YOU DON'T WANT TO TRACK

Example:

- Content of `.gitignore` file:

```
# you can use comments too!
__pycache__/  # this ignores a whole directory
*.bak         # name of backup files
*~            # some editors create backup files ending with '~'
```

- Let's create an annoying file:

```
$ touch .DS_Store  # annoying meta-data file in Mac OSX
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .DS_Store

nothing added to commit but untracked files present (use "git add" to track)
```

IGNORE DATA THAT YOU DON'T WANT TO TRACK

Example:

- We can add it to the `.gitignore` file:

```
$ echo '.DS_Store' >> .gitignore
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .gitignore

no changes added to commit (use "git add" and/or "git commit -a")
```

- Notice that `.DS_Store` does no longer show up as an *untracked* file.
- Of course we have *modified* our tracked `.gitignore` file, which we must now stage and commit to make the changes persistent.

A FEW COMMENTS ON `.gitignore`

- **It is essential that you keep your repository clean**, the `.gitignore` file is the key to a clean house.
- You usually have one in the root directory and possibly others in more specific locations of your project
- Often they have specific entries for the programming language you are using. E.g. for `python` you want to ignore the `__pycache__` directories.
- GitHub offers you some templates for this file at the time when you create a new repository, have a look at them to get an idea. I often prefer to create them from scratch and extend them incrementally.

A FEW COMMENTS ON `.gitignore`

- `git` is great for VCS of *text* files
- It can handle binary files but they are more difficult to track. This often leads to bloating up the size of your `.git` repository.
- Ignore such files in your `.gitignore` files. E.g. for PDF files:

  ```
  *.pdf # ignore all pdf's
  ```

  If you *must* make an exception, force add the file to the index:

  ```
  $ git add --force important.pdf
  ```

- You can use " `!` " as ***negation*** in your `.gitignore` files:

  ```
  *     # ignore everything (remember the wildcard in the shell, it expands to anything)
  !*.py # except any file with the .py suffix
  ```

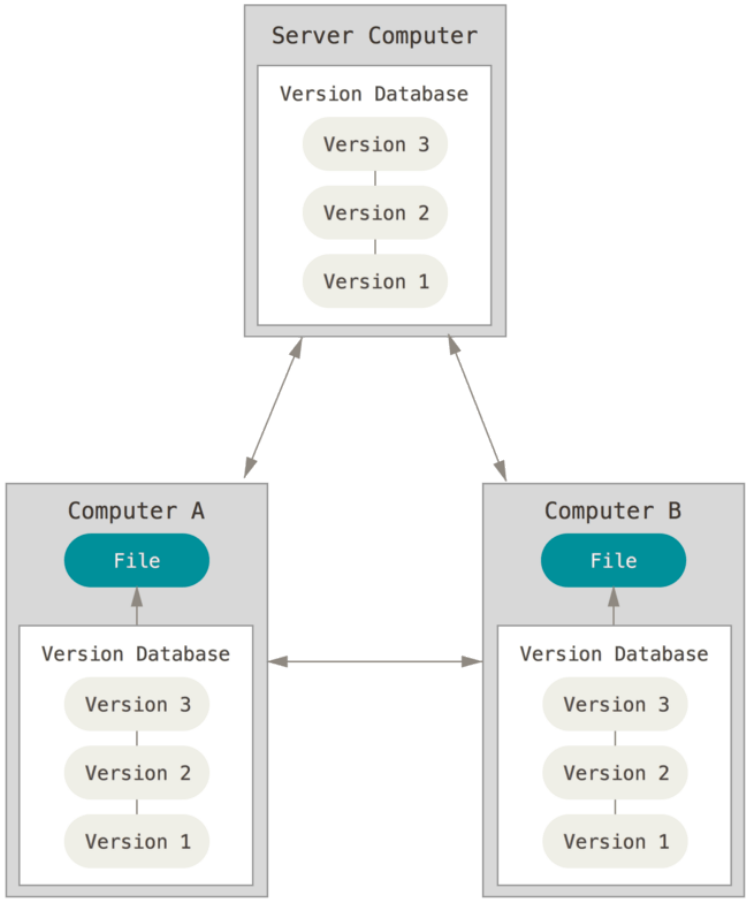
GETTING `git help`

- All commands in `git` have proper manual pages
- You can get them in two ways:
  1. Via `git`: `git help commit`
  2. Via `man`: `man git-commit`

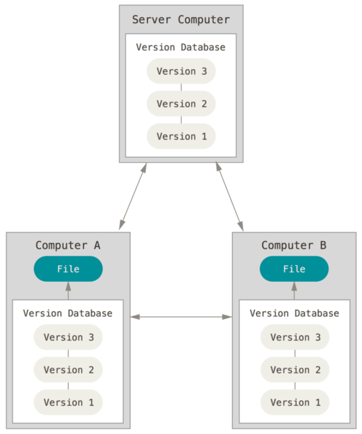
REMOTE REPOSITORIES

REMOTE REPOSITORIES

Recall distributed VCS:



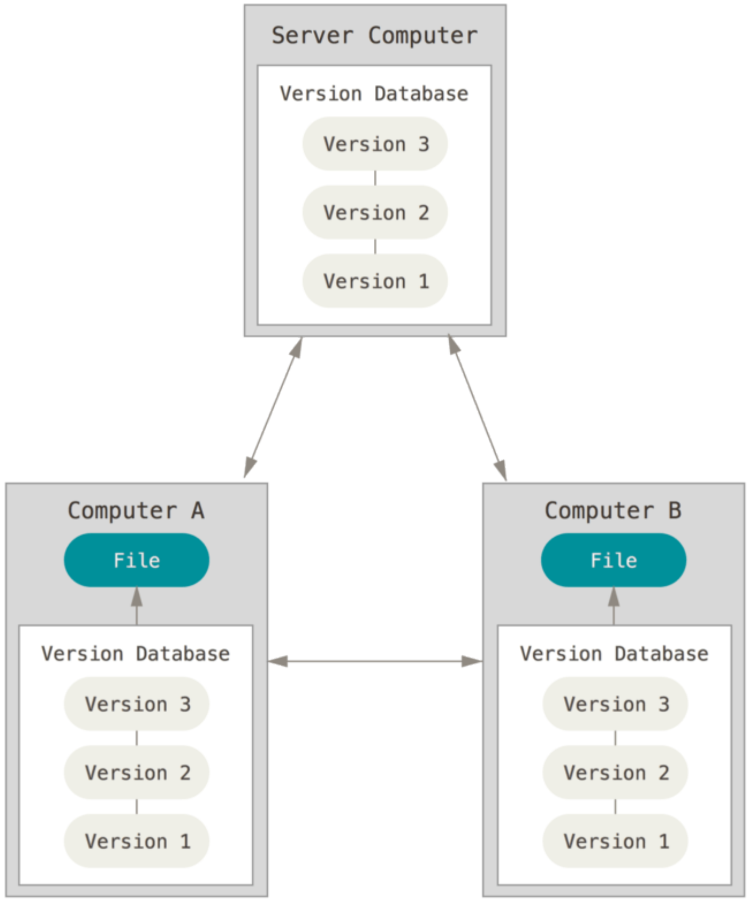
REMOTE REPOSITORIES



- The "server computer" is called a *remote*.
- It can be a server from GitHub, for example, but it can also be local on your computer.
- The term "remote" does not necessarily imply that it is some place else on the internet, only that the remote repository is *somewhere* else.
- All that a remote repository needs is the content of the `.git` directory, such a repository is called a ***bare*** repository.
- You can list the remotes with `git remote show`.

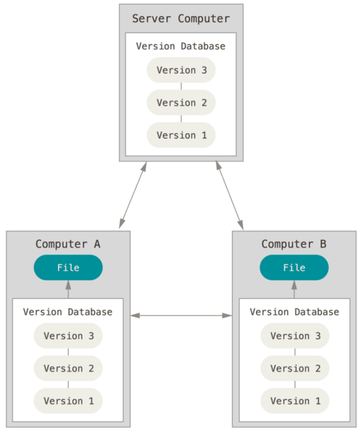
REMOTE REPOSITORIES DEMO

Let us emulate these `git` repositories:



REMOTE REPOSITORIES DEMO

Let us emulate these `git` repositories:



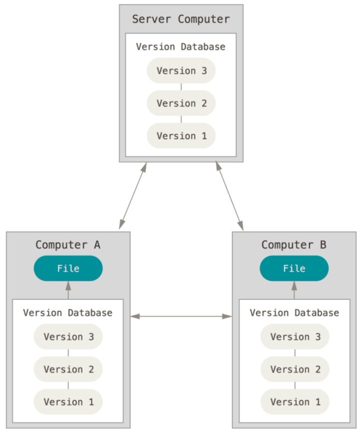
- ***As soon as you work with collaborators***, you may run into `git` *rejecting* your push because your local repository is not up to date with the remote you are trying to push to:

```
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to '../remote/'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

- Simply fetch and merge these changes first before you push. ***Recall:*** `git pull` does both of these tasks in one shot.

REMOTE REPOSITORIES DEMO

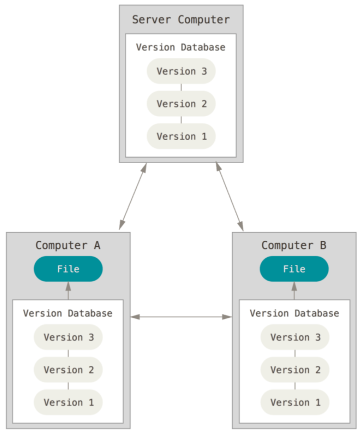
Let us emulate these `git` repositories:



- The pull in the previous demo went smooth. In the real world, this may not always be the case.
- In many cases `git` is very good at performing merges...
- ...but what should `git` do when two collaborators modify the *exact same* place in a file?
- Let's continue our demo and see...

REMOTE REPOSITORIES DEMO

Let us emulate these `git` repositories:



- *Merge conflicts* are not uncommon and are possibly the trickiest part in `git`. They are hard to avoid in distributed VCS.
- And are another good reason why you want to keep your individual commits small and well-structured.
- `git` offers you a tool to resolve merge conflicts: `git help mergetool`

REMOTE REPOSITORIES

- You can add as many remotes as you like.
- If you do not setup a tracking branch for `git push`, then you must explicitly say which remote you want to push to and which branch.
- The same is true for `git pull`.
- On GitHub you can choose to use `https` or `ssh` to communicate with a remote.
- If you use `https` you now must create a *token* in "Settings/Developer Settings".
- For `ssh` you should generate an `ssh` key using `ssh-keygen -t rsa -b 4096` and upload the *public* key part to GitHub.

BRANCHING IN `git`

BRANCHING IN `git`

- We have encountered branches already but not said much about them up to now.
- In your GitHub repo, `main` or `master` *are branches*. In `git` *everything* is a branch.
- Branches are your ***main tool*** for development. They allow you to get messy without breaking anything in the main branch and you can just discard them and start over.
- Historically, branching is an expensive task in VCS, ***not so in*** ***`git`***
- In `git`, a branch is just an alias name for a SHA-1 hash that *points* to a particular snapshot

```
$ cat .git/refs/heads/master
d5278fbd2931b75e9f41ef031448fa1b2696fce4
```

BRANCHING IN `git`

- Assume you have a new repository and you just created the initial commit `A`:

  ```
  A master
  ```

  The "pointer" named `master` contains the same SHA-1 entry as the one generated for commit `A`.
- Now suppose we make two more commits `B` and `C`. The pointer that describes the `master` branch moves along:

  ```
  A---B---C master
  ```

- At this point in your development stage, you notice a strange behavior of your code and you suspect that a bug has been introduced. ***How to proceed now?***

BRANCHING IN `git`

- You could continue on `master` but this *is not a good idea* because fixing a bug requires you to throw things around. *Create a new branch for this bug fix:*

```
$ git switch -c bugfix1  # the -c option creates the branch if it does not exist
```

Note: in older versions of `git` a new branch was checked out like this

```
$ git checkout -b bugfix1  # the -b option creates the branch if it does not exist
```

The reason this is confusing is because `git checkout` has *dual* meaning:

1. It can checkout branches
2. It can checkout individual files and restore their content

Newer versions of `git` split these tasks by introducing two new commands:

1. `git switch`: switch branches
2. `git restore`: restore files

- Our revision timeline now looks like this:

```
A---B---C master   # branch point is C
         \
          bugfix1  # branch point is C, active branch and what HEAD points to
```

BRANCHING IN `git`

- Our revision timeline now looks like this:

```
A---B---C master    # branch point is C
         \
          bugfix1   # branch point is C, active branch and what HEAD points to
```

There is a special pointer in `git` called `HEAD`. It always points to the currently active branch.

- Now we do some work and fix this bug. Assume the next two commits `D` and `E` implement these fixes:

```
A---B---C master   # this is the active branch now
         \
          D---E bugfix1   # this branch contains the bug fixing code
```

We also switched back to the `master` branch.

Which `git` command did we use for this? `git switch master`

BRANCHING IN `git`

- We have now tested our changes on the `bugfix1` branch and are back on the `master` branch as we would like to ***merge*** the history of `bugfix1` into `master`.

```
A---B---C master   # this is the active branch now
         \
          D---E bugfix1   # this branch contains the bug fixing code
```

- Because there are no new commits on `master` since we branched off, the merge is trivial. `git` has two options:

  1. Fold `bugfix1` and `master` together (*fast-forward merge*)
  2. Create a *merge commit* which joins `bugfix1` and `master` in a common commit.

BRANCHING IN `git`

- Situation *before* merge:

```
A---B---C master  # this is the active branch now
         \
          D---E bugfix1  # this branch contains the bug fixing code
```

- ***Fast-Forward merge***: this is the default that `git` assumes. Assume you are on the `master` branch, the command for this merge is
  `git merge [--ff] bugfix1` (the fast-forward option `--ff` is implied if not given)
  After this merge your history looks like this:

```
A---B---C---D---E master
                 \
                  bugfix1  # this branch is now dangling
```

  The `bugfix1` branch is now *fully* merged in `master`. We do not need it anymore and it is good practice to remove it:
  `git branch -d bugfix1`

```
A---B---C---D---E master
```

BRANCHING IN `git`

- Situation *before* merge:

```
A---B---C master   # this is the active branch now
         \
          D---E bugfix1   # this branch contains the bug fixing code
```

- ***Non Fast-Forward merge***: this type of merge creates a common join commit for the merge:

`git merge --no-ff bugfix1`

After this merge your history looks like this:

```
A---B---C-------F master # F is called a merge commit
         \     /
          D---E bugfix1   # this branch is now dangling
```

Same story for keeping your repository clean:

`git branch -d bugfix1`

```
A---B---C-------F master
         \     /
          D---E
```

BRANCHING IN `git`

Compare the difference of the two approaches:

FAST-FORWARD

```
$ git log --oneline --graph
* 57f4883 (HEAD -> master) Commit E
* d5278fb Commit D
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

Linear history:

```
A---B---C---D---E master
```

WITH MERGE-COMMIT

```
$ git log --oneline --graph
*   4466977 (HEAD -> master) Merge branch 'b
|\                           (Commit F)
| * 57f4883 Commit E
| * d5278fb Commit D
|/
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

Non-linear history:

```
A---B---C-------F master
         \     /
          D---E
```

BRANCHING IN `git`

Compare the difference of the two approaches:

FAST-FORWARD

```
$ git log --oneline --graph
* 57f4883 (HEAD -> master) Commit E
* d5278fb Commit D
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

WITH MERGE-COMMIT

```
$ git log --oneline --graph
*   4466977 (HEAD -> master) Merge branch 'b
|\                           (Commit F)
| * 57f4883 Commit E
| * d5278fb Commit D
|/
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

- Some people argue that creating merge commits adds *noise* to your history (technically they are not needed)
- They preserve your branching history, which may be useful to understand at some point in your project(s)
- Some projects have policies for that, be aware of it

STASHING CHANGES WITHOUT COMMITTING

Assume you find yourself in this situation:

```
A---B---C master
         \
          D-* bugfix1 # a bugfix branch, active branch ('*' means modified files)
```

Common scenario: you stop work on the `bugfix1` branch temporarily and need to switch to some other branch (say `master`). Your work on `bugfix1` is not ready to be committed yet.

```
$ git status
On branch bugfix1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file

no changes added to commit (use "git add" and/or "git commit -a")
$ git switch master
error: Your local changes to the following files would be overwritten by checkout:
        file
Please commit your changes or stash them before you switch branches.
Aborting
```

STASHING CHANGES WITHOUT COMMITTING

Assume you find yourself in this situation:

```
A---B---C master
         \
          D-* bugfix1 # a bugfix branch, active branch ('*' means modified files)
```

Common scenario: you stop work on the `bugfix1` branch temporarily and need to switch to some other branch (say `master`). Your work on `bugfix1` is not ready to be committed yet.

If committing is too early, you can use `git stash` to *temporarily* stash your changes:

```
$ git stash  # push current work on top of stash stack
Saved working directory and index state WIP on bugfix1: b955cbe Adding new file on bugfi
$ git stash list  # list all stashed work, WIP means Work In Progress
stash@{0}: WIP on bugfix1: b955cbe Adding new file on bugfix1
$ git status
On branch bugfix1
nothing to commit, working tree clean
$ git switch master  # do some other work on master
Switched to branch 'master'
```

STASHING CHANGES WITHOUT COMMITTING

Assume you find yourself in this situation:

```
A---B---C master
         \
          D-* bugfix1 # a bugfix branch, active branch ('*' means modified files)
```

Common scenario: you stop work on the `bugfix1` branch temporarily and need to switch to some other branch (say `master`). Your work on `bugfix1` is not ready to be committed yet.

If committing is too early, you can use `git stash` to *temporarily* stash your changes:

```
$ git switch bugfix1  # when done return to your bugfix1 branch
Switched to branch 'bugfix1'
$ git stash pop  # apply your last stashed changes, i.e. stash@{0}
On branch bugfix1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (e3e552a02d84049a314e77edcb34dae0987ef145)
```

LINEAR HISTROY AND `git rebase`

Lets return to our previous state but now we have a collaborator who did work on `master` in the meantime (the commit labels `A`, `B`,... are only symbolic, don't take them literally):

```
A---B---C---D---E master   # work has advanced on this branch
         \
          F---G bugfix1    # this branch contains the bug fix, active branch
```

Can you apply a fast-forward merge strategy in this case?

LINEAR HISTROY AND `git rebase`

```
A---B---C---D---E master   # work has advanced on this branch
         \
          F---G bugfix1    # this branch contains the bug fix, active branch
```

Can you apply a fast-forward merge strategy in this case?

- **You can not.** Remember, once a history is recorded by computing the SHA-1 hash, we can not change it anymore.
- There are two options:
  1. Merge via a merge commit (same as before)
  2. *If* `bugfix1` is a branch that only exists in your local `.git` repository, we can rebase and therefore *rewrite the local history* that none of your collaborators has seen yet.

LINEAR HISTROY AND `git rebase`

```
A---B---C---D---E master   # work has advanced on this branch
         \
          F---G bugfix1    # this branch contains the bug fix, active branch
```

- ***Merge via merge commit:*** same as in the previous case where work on `master` did not advance:
  1. `git switch master` (change to the target branch)
  2. `git merge bugfix1`

     ```
     A---B---C---D---E---H master   # this is the active branch now
              \         /
               F-------G bugfix1    # this branch contains the bug fix, now dangling
     ```

  3. `git branch -d bugfix1` (clean up)

LINEAR HISTROY AND `git rebase`

```
A---B---C---D---E master   # work has advanced on this branch
         \
          F---G bugfix1    # this branch contains the bug fix, active branch
```

- ***Rebase and merge:*** here we first *rebase* our `bugfix1` branch *onto* the advanced `master` branch and then use a fast-forward merge to *linearize* the history (we start with the state shown above):
  1. `git rebase master` (Rewriting history here!)
     ```
     A---B---C---D---E master   # work has advanced on this branch
                      \
                       F'---G' bugfix1   # after rebase, active branch
     ```
     - Commits `F'` and `G'` have a ***different*** SHA-1 than `F` and `G`, therefore, history is rewritten!
     - Their *time stamp* remains the same
  2. `git switch master && git merge bugfix1`
     ```
     A---B---C---D---E---F'---G' master   # rebased history
                              \
                               bugfix1   # now dangling
     ```
  3. `git branch -d bugfix1` (clean up)

LINEAR HISTROY AND `git rebase`

- `git rebase` unwinds commits and re-applies them on top of another commit. Naturally, this changes your history
- Rebased histories can have commit time stamps that are *not* in chronological order
- But rebasing allows to *linearize* your history
- Some projects have policies for these merge strategies. Be sure to check them out before collaborating.

You can get into ***big trouble*** if you rebase a history and (forcefully) push it to a remote where others can pull from. `git` does not allow you to do this by default, but you can force it with `git push --force`. It will *invalidate* the history in all your collaborators' local repositories. Be careful.

RECAP

- *More `git` basics:* use `git status` often.
- *Remote repositories:* differences between a normal `.git` repository and a *bare* `.git` repository, merge conflicts.
- *Branching in `git`:* fast-forward merges and rebasing, linear and non-linear histories.

REFERENCE

- Pro Git book by Scott Chacon and Ben Straub, Apress (Chapters 2, 3 and 4)
- `git` command reference