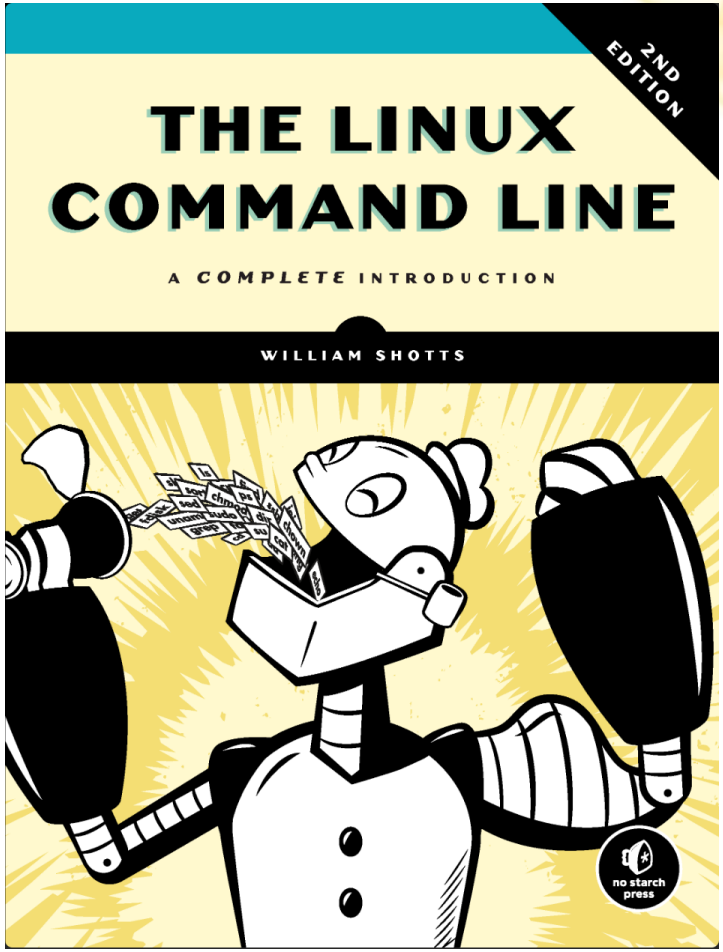
UMD DATA605 - Big Data Systems

Git, Data Pipelines

Instructor: Dr. GP Saggese - gsaggese@umd.edu

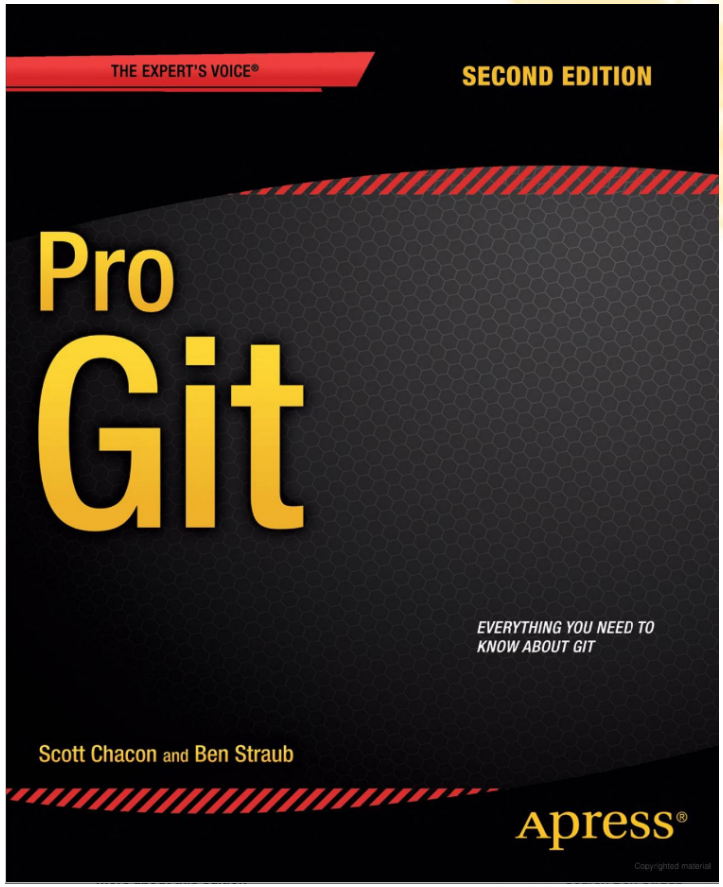
Bash / Linux: Resources

- Command line
  - Command-Line for Beginners
  - E.g., `find`, `xargs`, `chmod`, `chown`, symbolic, and hard links
- How Linux works
  - Processes
  - File ownership and permissions
  - Virtual memory
  - How to administer a Linux box as root
- Mastery
  - The Linux Command Line



Git Resources

- Concepts in the slides
- Tutorial: Tutorial Git
- We will use Git during the project
- Mastery: Pro Git (free)
- Web resources:
  - https://githowto.com
  - dangitgit.com (without swearing)
  - Oh Sh*t, Git!?! (with swearing)
- Playgrounds
  - https://learngitbranching.js.org



Oh Shit, Git!?!

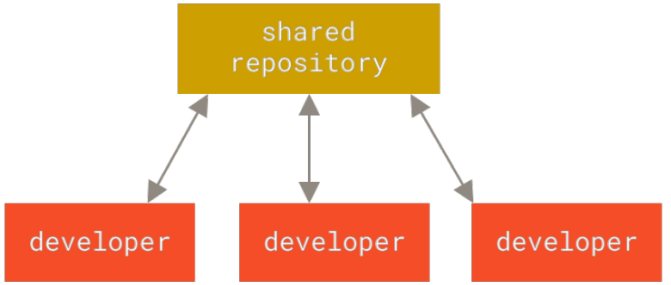
Git is hard: screwing up is easy, and figuring out how to fix your mistakes is fucking impossible. Git documentation has this chicken and egg problem where you can't search for how to get yourself out of a mess, *unless you already know the name of the thing you need to know about* in order to fix your problem

Version Control Systems (1/2)

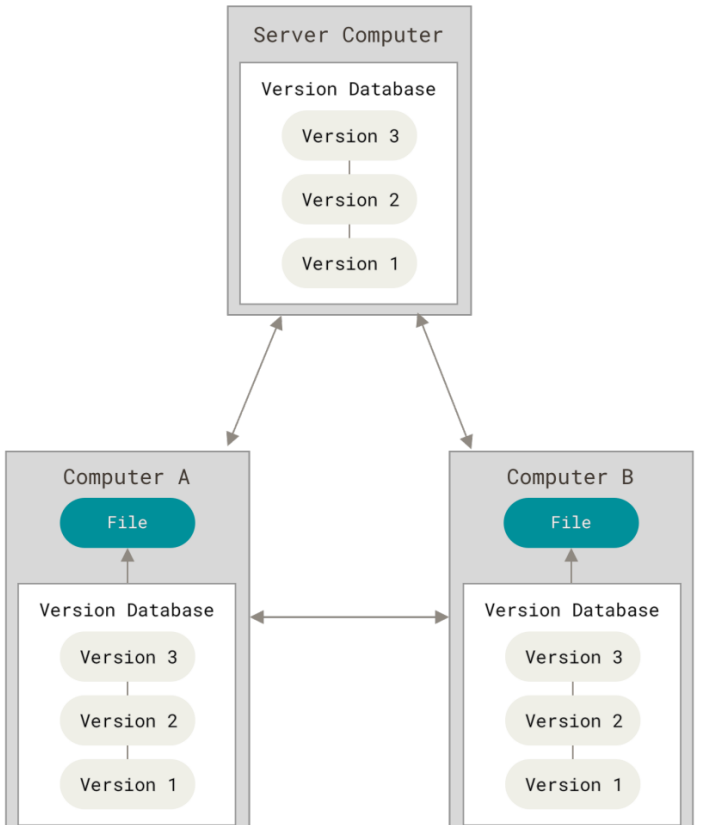
- A **Version Control System** (VCS) is a system that allows to:
  - Record changes to files
  - Recall specific versions later (like a "file time-machine")
  - Compare changes to files over time
  - Track *who* changed *what* and *when* and *why*
- **Simplest "VCS"**
  - Make a copy of a dir and add
    - `_v1` (bad); or
    - a timestamp `_20220101` (better)
  - **Cons**: It kind of works for one person, but doesn't scale

Version Control Systems (2/2)

- **Centralized VCS**
  - E.g., `Perforce`, `Subversion`
  - A server stores the code, clients connect to it
  - **Cons**: If the server is down, nobody can work

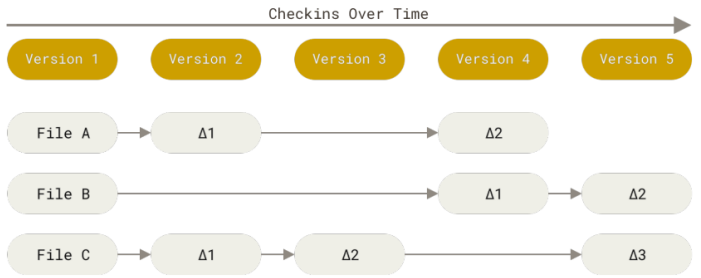


- **Distributed VCS**
  - E.g., `Git`, `Mercurial`, `Bazaar`, `Dart`
  - Each client has the entire history of the repo locally
  - Each node is both a client and a server
  - **Cons**: complex

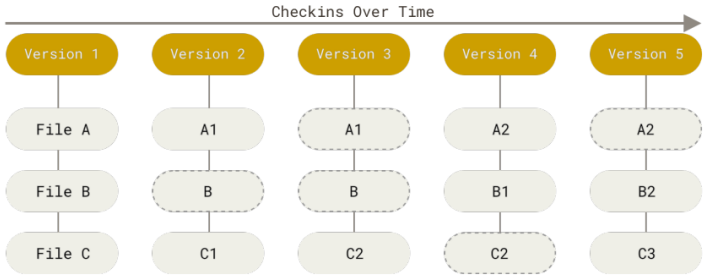


VCS: How to Track Data

- Consider a directory with project files inside
- **How do you track changes to the data?**
- **Delta-based VCS**
  - E.g., `Subversion`
  - Store the data in terms of patches (changes of files over time)
  - Can reconstruct the state of the repo by applying the patches



- **Stream of snapshots VCS**
  - E.g., `Git`
  - Store data in terms of snapshots of a filesystem
  - Take a "picture" of what files look like
  - Store reference (hash) to the snapshots
  - Save link to previous identical files

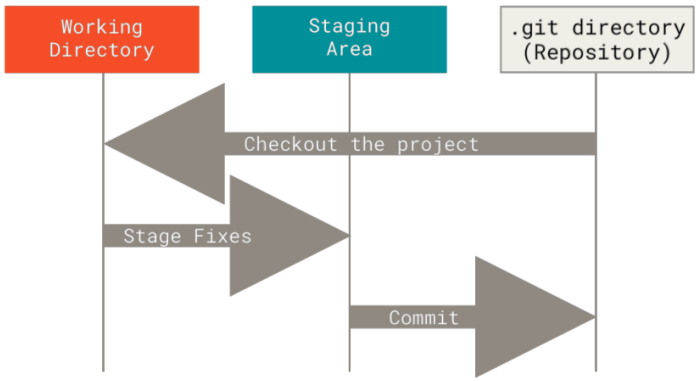


Git

- **Almost everything is local for Git**
  - History stored locally in each node
  - Diff-ing files done locally
    - Centralized VCS requires server access
  - Commit to local copy
    - Upload changes with network connection
- **Almost everything is undoable in Git**
  - No data corruption
    - Everything checksummed
  - Nothing lost
  - Disclaimer:
    - Commit (at least locally) or stash
    - Know how to do it to avoid "git hell"
- **Git is a mini key-value store with a VCS built on top**
  - Exactly true
  - Two layers:
    - "porcelain": key-value store for file-system
    - "plumbing": VCS layer

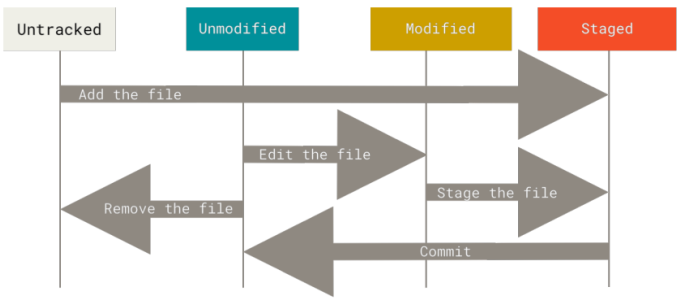
Sections of a Git Project

- There are 3 main sections of a Git project
  - **Working tree** (aka checkout)
    - Version of code on the filesystem for use and modification
  - **Staging area** (aka cache, index)
    - File in `.git` storing info for the next commit
  - **Git directory** (aka .git)
    - Stores metadata and objects (like a DB)
    - The repo itself with all history
    - Cloning gets you the project's `.git`



States of a File in Git

- Each file can be in 4 states from Git point-of-view
  - **Untracked**: files not under Git version control
  - **Modified**: changed files, not committed yet
  - **Staged**: marked modified files for next commit
  - **Committed**: data stored in local DB



Git Tutorial

- **Git tutorial on class repo**
  - Follow the `README`
- **How to use a tutorial**
  - Type commands one-by-one
    - Avoid copy-paste
  - Observe results
    - Understand each line
  - Experiment
    - *"What happens if I do this?"*
    - *"Does the result match my mental model?"*
  - Learn command line before GUI
    - GUIs hide details and you become dependent on it
- Go through **recommended Git book** and try all examples
  - Use online tutorials
- Build your own **cheat sheet**
  - Reuse others' cheat sheets only if familiar
- Achieve **mastery of basic tools**
  - Bash, Git, editor
  - Python, Pandas
  - ...

Git: Daily Use

- Check out a project (`git clone`) or start from scratch (`git init`)
  - Only once per Git project client
- **Daily routine**
  - Modify files in working tree (`vi ...`)
  - Add files (`git add ...`)
  - Stage changes for next commit (`git add -u ...`)
  - Commit changes to `.git` (`git commit`)
- **Use a branch to group commits together**
  - Isolate code from changes in `master`
  - Merge `master` into branch
  - Isolate `master` from changes
  - Pull Request (PR) for code review
  - Merge PR into upstream

Git Remote

- **Remote repos**: versions of the project hosted online
  - Manage remote repos to collaborate
  - Push/pull changes
  - `git remote -v`: show remotes
  - `git fetch`: pull data from remote repo you don't have locally
  - `git pull`: shorthand for `git fetch origin` + `git merge master --rebase`
  - `git push <REMOTE> <BRANCH>`: push local data to remote
    - E.g., `git push origin master`
- Multiple forks of the same repo with different policies
  - E.g., read-only, read-write
- If someone pushed to remote, you **can't push changes immediately**:
  - Fetch changes
  - Merge changes to your branch
  - Resolve conflicts, if needed
  - Test project sanity (e.g., run unit tests)
  - Push changes to remote

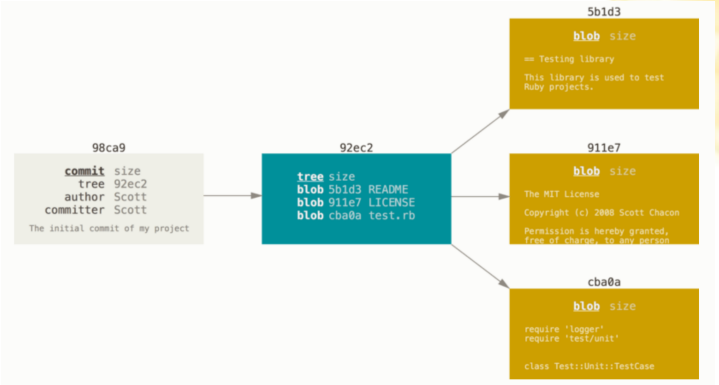
Git Tagging

- Git allows marking points in history with a **tag**
  - E.g., release points
  - Check out a tag
- Enter detached `HEAD` state
  - Committing won't add changes to the tag or branch
  - Commit will be "unreachable," reachable only by commit hash

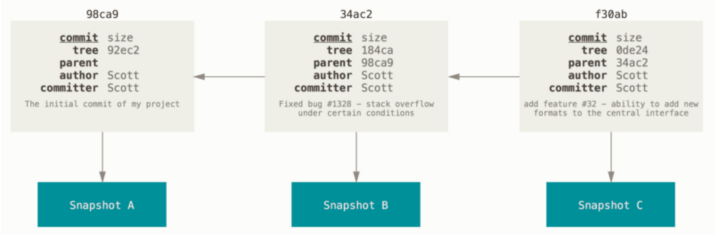
Git Internals

- **Understand Git only if you understand its data model**
  - Git is a key-value store with a VCS interface
  - Key = hash of a file
  - Value = content of a file
- **Git objects**
  - Commits: pointers to the tree and commit metadata
  - Trees: directories and mapping between files and blobs
  - Blobs: content of files
- Refs:
  - Easy: Understanding Git Data Model
  - Hard-core: Git internals

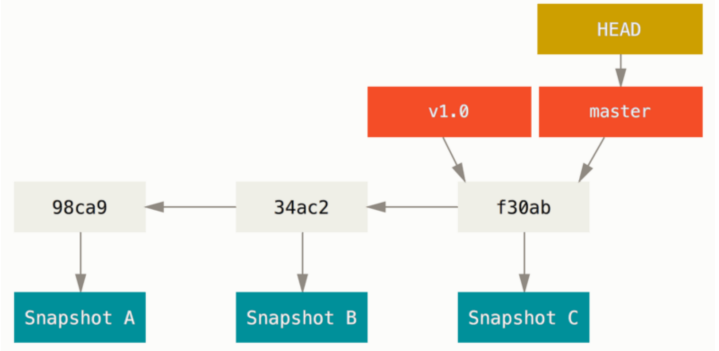
Commit tree



Commit parents

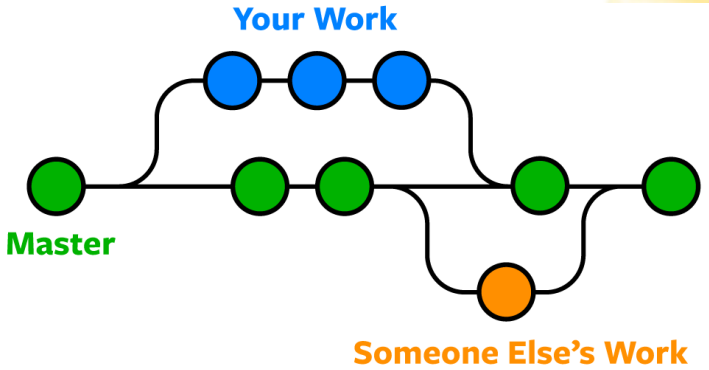


Commit history of a branch



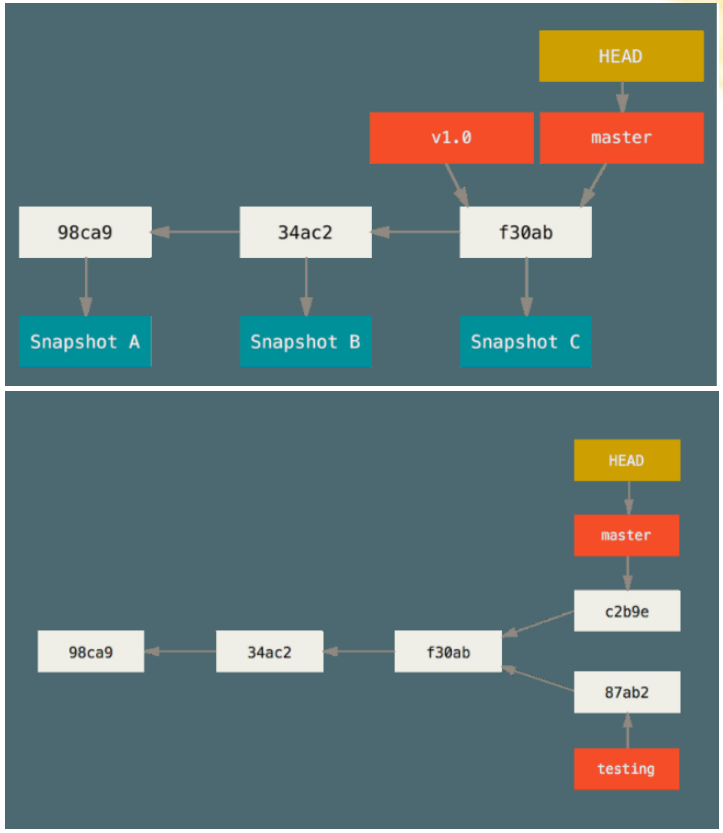
Git Branching

- **Branching**
  - Diverge from main development line
- **Why branch?**
  - Work without affecting main code
  - Avoid changes in main branch
  - Merge code downstream for updates
  - Merge code upstream after completion
- **Git branching is lightweight**
  - Instantaneous
  - Branch is a pointer to a commit
  - Git stores data as snapshots, not file differences
- **Git workflows branch and merge often**
  - Multiple times a day
  - Surprising for users of distributed VCS
    - E.g., branch before lunch
  - Branches are cheap
    - Use them to isolate and organize work



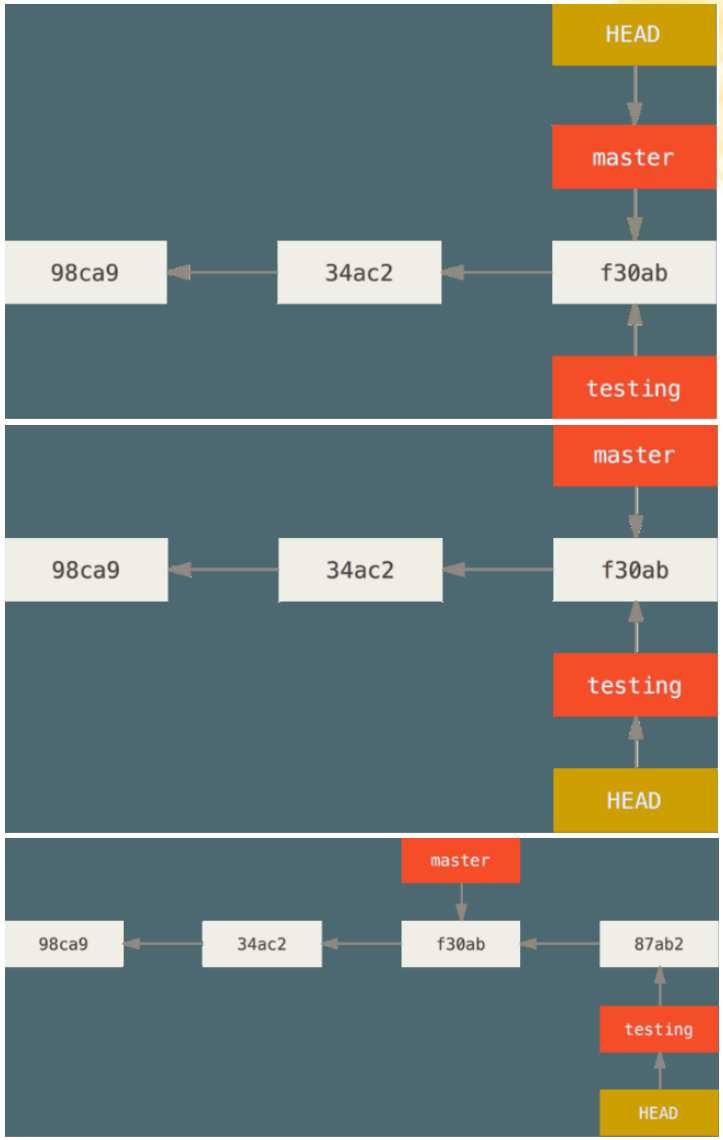
Git Branching

- `master` (or `main`) is a normal branch
  - Pointer to the last commit
  - Moves forward with each commit
- `HEAD`
  - Pointer to the local branch
  - E.g., `master`, `testing`
  - `git checkout <BRANCH>` moves across branches
- `git branch testing`
  - Create a new pointer `testing`
  - Points to the current commit
  - Pointer is movable
- Divergent history
  - Work progresses in two "split" branches



Git Checkout

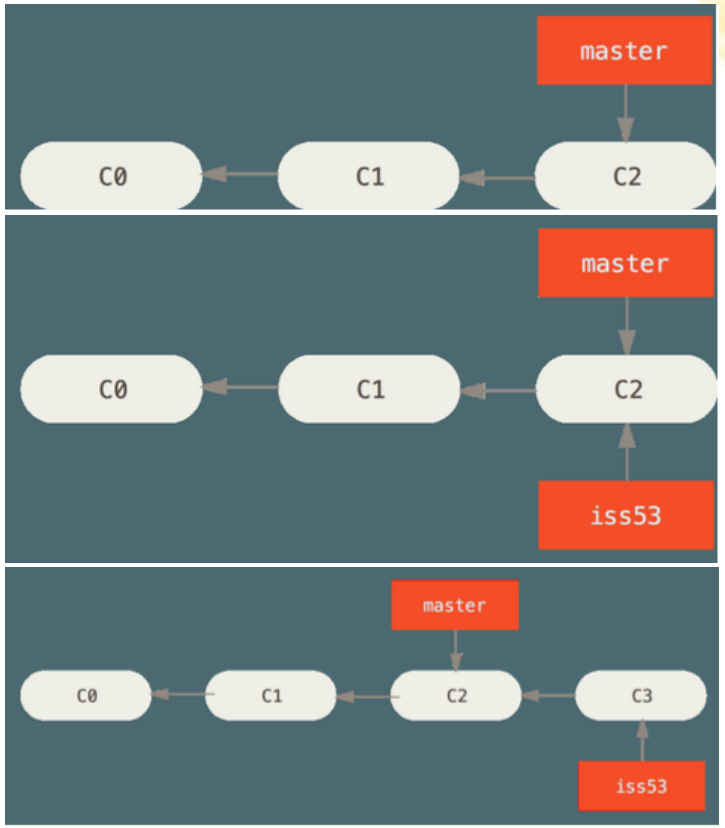
- `git checkout` switches branch
  - Move `HEAD` pointer to new branch
  - Change files in working dir to match branch pointer
- E.g., two branches, `master` and `testing`
  - You are on `master`
  - `git checkout testing`
  - Pointer moves, working dir changes
  - Keep working and commit on `testing`
  - Pointer to `testing` moves forward



Git Branching and Merging

- Tutorials
  - Work on main
  - Hot fix
- Start from a project with some commits
- Branch to work on a new feature "Issue 53"

```
> git checkout -b iss53
work ... work ... work
> git commit
```

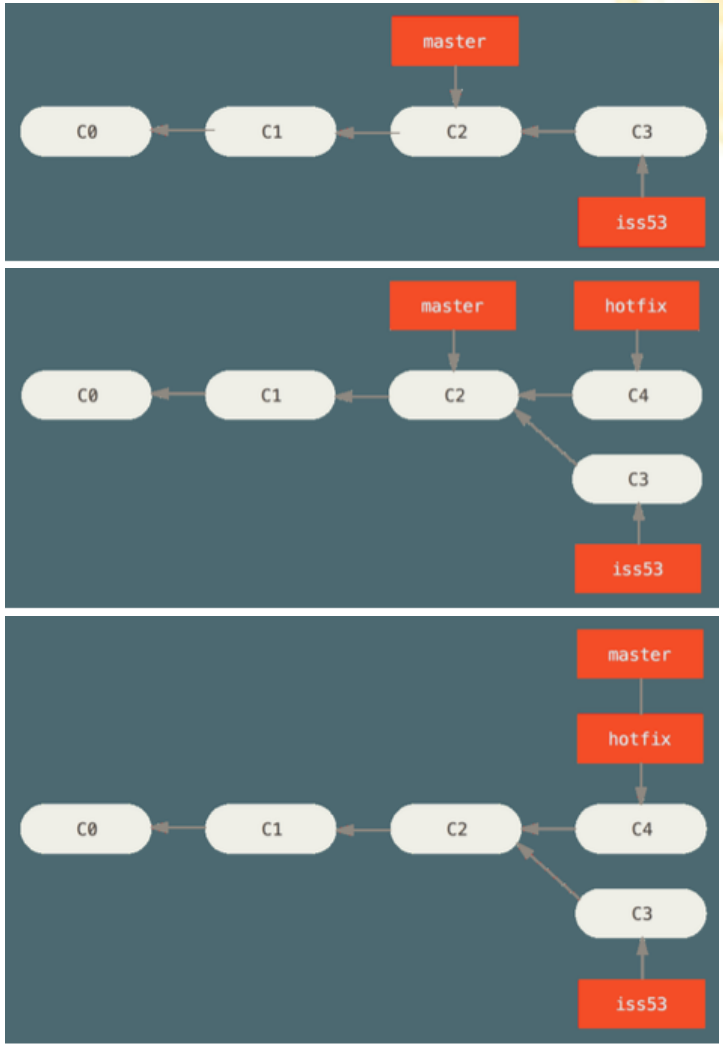


Git Branching and Merging

- **Need a hotfix** to `master`

```
> git checkout master
> git checkout -b hotfix
fix ... fix ... fix
> git commit -am "Hot fix"
> git checkout master
> git merge hotfix
```

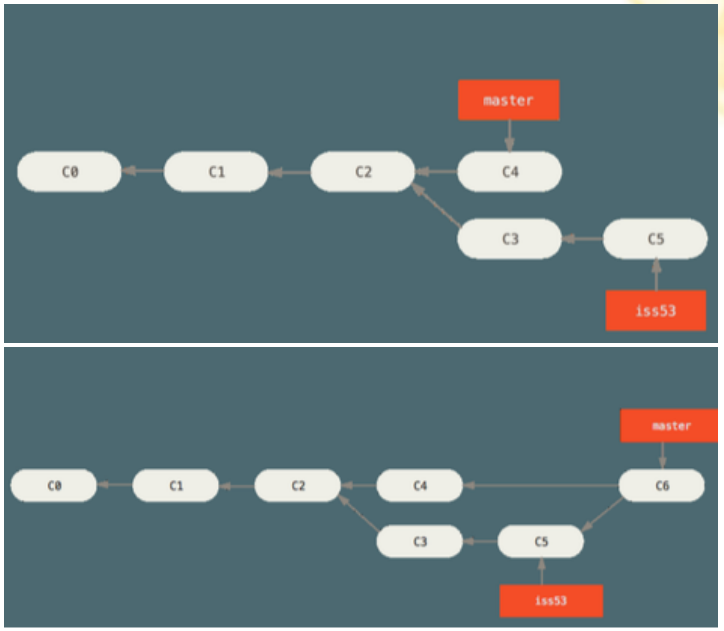
- **Fast forward**
  - Now there is a divergent history between `master` and `iss53`



Git Branching and Merging

- Keep working on `iss53`
  `> git checkout iss53`
  `work ... work ... work`
  - The branch keeps diverging
- At some point you are done with `iss53`
  - You want to merge your work back to `master`
  - Go to the target branch

  `> git checkout master`
  `> git merge iss53`
- Git can't fast forward

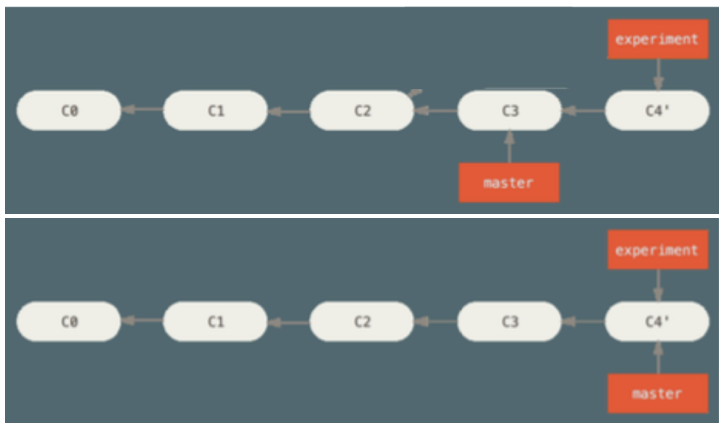


- Git creates a new snapshot with the 3-way "merge commit" (i.e., a commit with more than one parent)
- Delete the branch `> git branch -d iss53`

Fast Forward Merge

- **Fast forward merge**
  - Merge a commit `X` with a commit `Y` that can be reached by following the history of commit `X`
- There is not divergent history to merge
  - Git simply moves the branch pointer forward from X to Y
- **Mental model**: a branch is just a pointer that says where the tip of the branch is
- E.g., `C4'` is reachable from `C3`

  ```
  > git checkout master
  > git merge experiment
  ```
- Git moves the pointer of master to `C4'`



Merging Conflicts

- Tutorial:
  - Merging conflicts
- Sometimes **Git can't merge**, e.g.,
  - The same file has been modified by both branches
  - One file was modified by one branch and deleted by another
- **Git**:
  - Does not create a merge commit
  - Pauses to let you resolve the conflict
  - Adds conflict resolution markers
- **User merges manually**
  - Edit the files `git mergetool`
  - `git add` to mark as resolved
  - `git commit`
  - Use PyCharm or VS Code

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

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

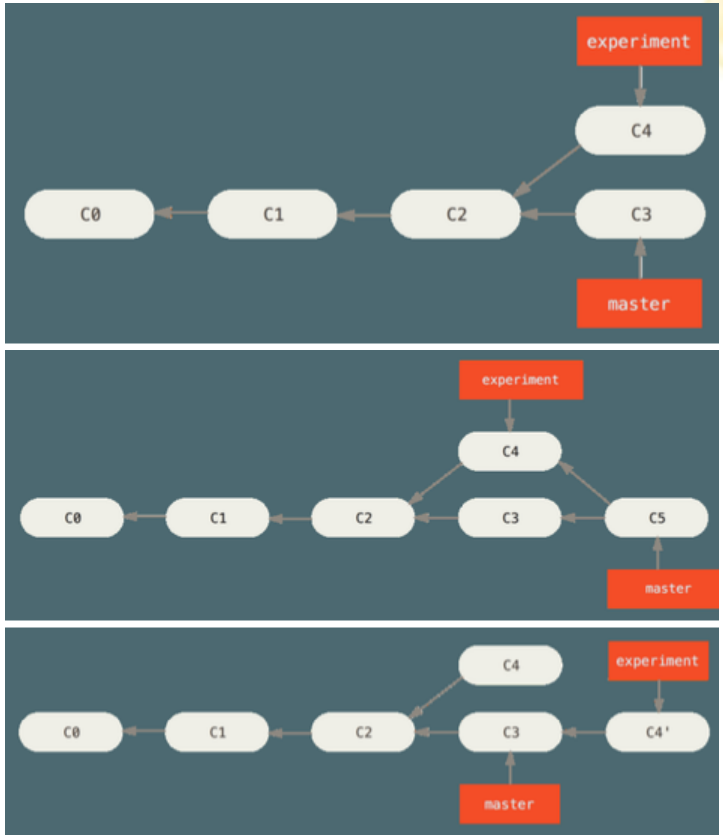
```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Git Rebasing

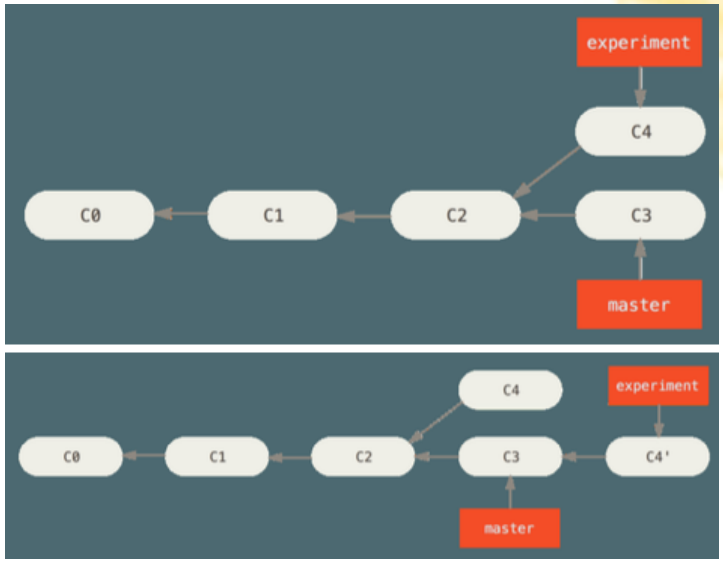
- In Git there are **two ways of merging divergent history**
  - E.g., `master` and `experiment` have a common ancestor `C2`
- **Merge**
  - Go to the target branch
    ```
    > git checkout master
    > git merge experiment
    ```
  - Create a new snapshot `C5` and commit
- **Rebase**
  - Go to the branch to rebase
    ```
    > git checkout experiment
    > git rebase master
    ```
  - Rebase algo:
    - Get all the changes committed in the branch (`C4`) where we are on (`experiment`) since the common ancestor (`C2`)
    - Sync to the branch that we are rebasing onto (`master` at `C3`)
    - Apply the changes `C4`
    - Only current branch is affected
    - Finally fast forward master



SCIENCE ACADEMY

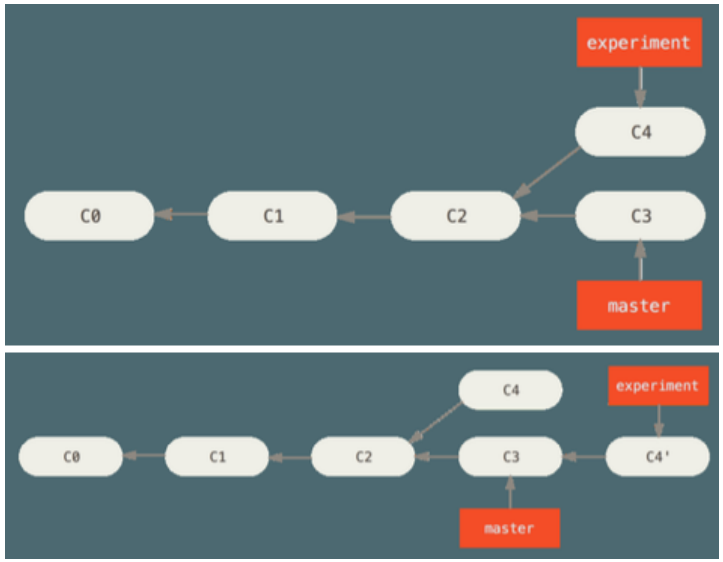
Uses of Rebase

- **Rebasing makes for a cleaner history**
  - The history looks like all the work happened in series
  - Although in reality it happened in parallel to the development in master
- **Rebasing to contribute to a project**
  - Developer
    - You are contributing to a project that you don't maintain
    - You work on your branch
    - When you are ready to integrate your work, rebase your work onto `origin/master`
  - The maintainer
    - Does not have to do any integration work
    - Does just a fast forward or a clean apply (no conflicts)



Golden Rule of Rebasing

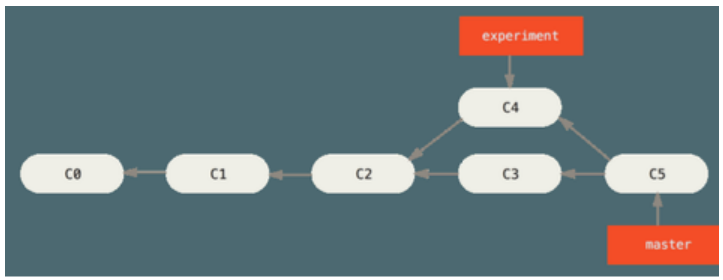
- **Remember**: rebasing means abandoning existing commits and creating new ones that are similar but different
- **Problem**
  - You push commits to a remote
  - Others pull commits and base work on them
  - You rewrite commits with `git rebase`
  - You push again with `git push --force`
  - Collaborators must re-merge work
- **Solution**
  - Strict: *"Do not ever rebase commits outside your repository"*
  - Loose: *"Rebase your branch if only you use it, even if pushed to a server"*



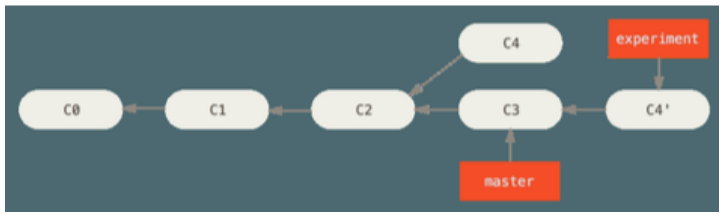
Rebase vs Merge: Philosophical Considerations

- Deciding **Rebase-vs-merge** depends on the answer to the question:
  - *What does the commit history of a repo mean?*

1. **History is the record of what actually happened**
   - *"History should not be tampered with, even if messy!"*
   - Use `git merge`



2. **History represents how a project should have been made**
   - *"You should tell the history in the way that is best for future readers"*
   - Use `git rebase` and filter-branch



Rebase vs Merge: Philosophical Considerations

- Many man-centuries have been wasted discussing rebase-vs-merge at the watercooler
  - Total waste of time! Tell people to get back to work!
- When you contribute to a project often people decide for you based on their preference
- **Best of the merge-vs-rebase approaches**
  - Rebase changes you've made in your local repo
    - Even if you have pushed but you know the branch is yours
    - Use `git pull --rebase` to clean up the history of your work
    - If the branch is shared with others then you need to definitively `git merge`
  - Only `git merge` to master to preserve the history of how something was built
- **Personally**
  - I like to squash-and-merge branches to `master`
  - Rarely my commits are "complete", are just checkpoints

Remote Branches

- **Remote branches** are pointers to branches in remote repos

```
git remote -v
origin  git@github.com:gpsaggese/umd_data605.git (fetch)
origin  git@github.com:gpsaggese/umd_data605.git (push)
```

- **Tracking branches**
  - Local references representing the state of the remote repo
  - E.g., `master` tracks `origin/master`
  - You can't change the remote branch (e.g., `origin/master`)
  - You can change tracking branch (e.g., `master`)
  - Git updates tracking branches when you do `git fetch origin` (or `git pull`)
- To share code in a local branch you need to push it to a remote

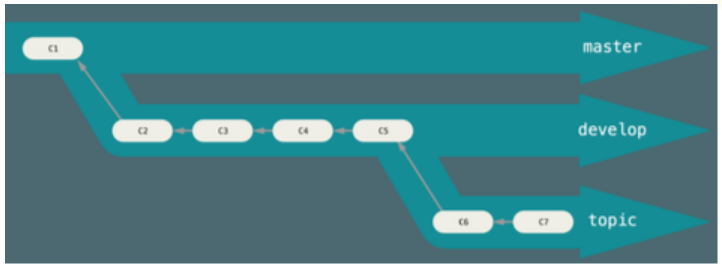
```
> git push origin serverfix
```

- To work on it

```
> git checkout -b serverfix origin/serverfix
```

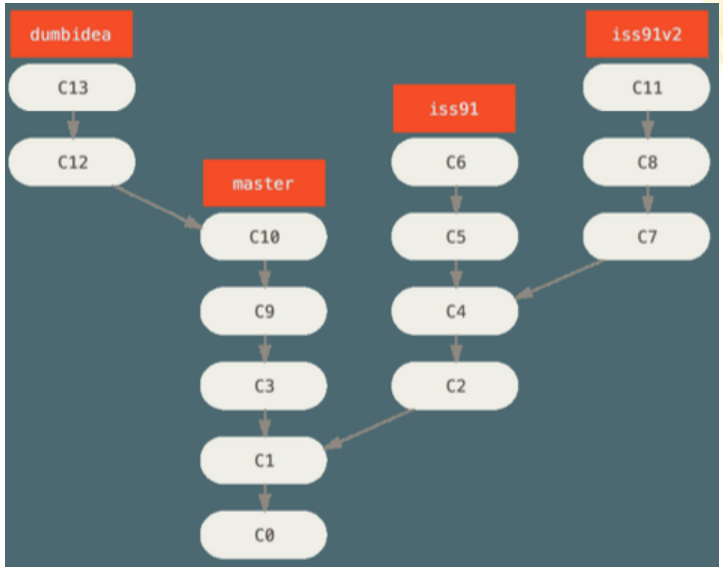
Git Workflows

- **Git workflows** = ways of working and collaborating using Git
- **Long-running branches** = branches at different level of stabilities, that are always open
  - `master` is always ready to be released
  - `develop` branch to develop in
  - topic / feature branches
  - When branches are "stable enough" they are merged up



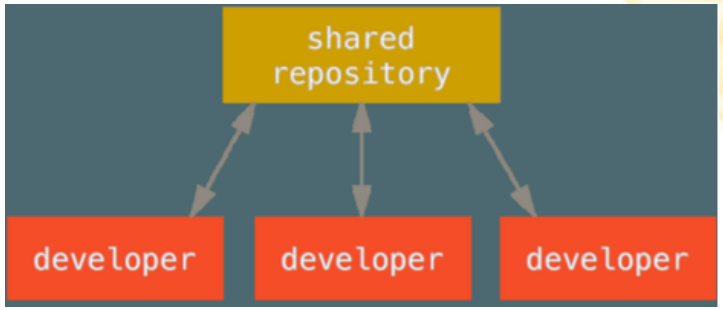
Git Workflows

- **Topic branches** = short-lived branches for a single feature
  - E.g., `hotfix`, `wip-XYZ`
  - Easy to review
  - Silo-ed from the rest
  - This is typical of Git since other VCS support for branches is not good enough
  - E.g.,
    - You start `iss91`, then you cancel some stuff, and go to `iss91v2`
    - Somebody starts `dumbidea` branch and merge to `master` (!)
    - You squash-and-merge your `iss91v2`



Centralized Workflow

- **Centralized workflow in centralized VCS**
  - Developers:
    - Check out the code from the central repo on their computer
    - Modify the code locally
    - Push it back to the central hub (assuming no conflicts with latest copy, otherwise they need to merge)
- **Centralized workflow in Git**
  - Developers:
    - Have push (i.e., write) access to the central repo
    - Need to fetch and then merge
    - Cannot push code that will overwrite each other code (only fast-forward changes)

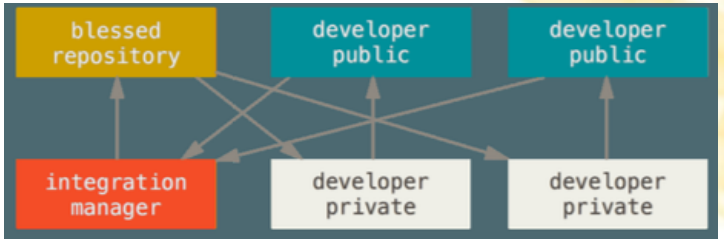


Forking Workflows

- Typically devs don't have permissions to update directly branches on a project
  - Read-write permissions for core contributors
  - Read-only for anybody else
- **Solution**
  - "Forking" a repo
  - External contributors
    - Clone the repo and create a branch with the work
    - Create a writable fork of the project
    - Push branches to fork
    - Prepare a PR with their work
  - Project maintainer
    - Reviews PRs
    - Accepts PRs
    - Integrates PRs
  - In practice it's the project maintainer that pulls the code when it's ready, instead of external contributors pushing the code
- **Aka "GitHub workflow"**
  - "Innovation" was forking (Fork me on GitHub!)
  - GitHub acquired by Microsoft for 7.5b USD

Integration-Manager Workflow

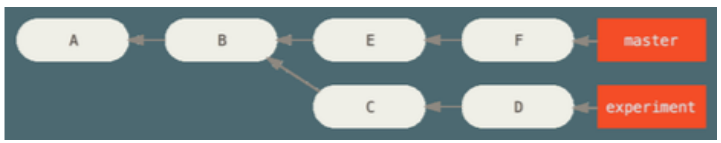
- This is the classical model for open-source development
  - E.g., Linux, GitHub (forking) workflow



1. **One repo is the official project**
   - Only the project maintainer pushes to the public repo
   - E.g., `causify-ai/csfy`
2. **Each contributor**
   - Has read access to everyone else's public repo
   - Forks the project into a private copy
     - Write access to their own public repo
     - E.g., `gpsaggese/csfy`
   - Makes changes
   - Pushes changes to his own public copy
   - Sends email to maintainer asking to pull changes (pull request)
3. **The maintainer**
   - Adds contributor repo as a remote
   - Merges the changes into a local branch
   - Tests changes locally
   - Pushes branch to the official repo

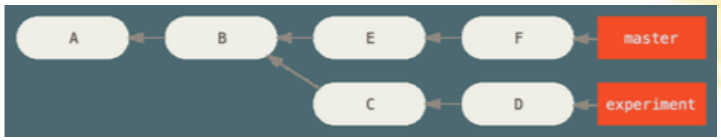
Git log

- `git log` reports info about commits
- **refs** are references to:
  - `HEAD` (commit you are working on, next commit)
  - `origin/master` (remote branch)
  - `experiment` (local branch)
  - `d921970` (commit)
- `^` after a reference resolves to the parent of that commit
  - `HEAD^` = commit before `HEAD`, i.e., last commit
  - `^2` means `^^`
  - A merge commit has multiple parents



Dot notation

- **Double-dot notation**
  - `1..2` = commits that are reachable from 2 but not from 1
  - Like a "difference"
  - `git log master..experiment` → `D,C`
  - `git log experiment..master` → `F,E`
- **Triple-dot notation**
  - `1...2` = commits that are reachable from either branch but not from both
  - Like "union excluding intersection"
  - `git log master...experiment` → `F,E,D,C`



Advanced Git

- **Stashing**
  - Copy state of your working dir (e.g., modified and staged files)
  - Save it in a stack
  - Apply later
- **Cherry-picking**
  - Rebase for a single commit
- **rerere**
  - = "Reuse Recorded Resolution"
  - Git caches how to solve certain conflicts
- **Submodules / subtrees**
  - Project including other Git projects

Advanced Git

- **bisect**
  - Bug appears at top of tree
  - Unknown revision where it started
  - Script returns 0 if good, non-0 if bad
  - `git bisect` finds revision where script changes from good to bad
- **filter-branch**
  - Rewrite repo history in a script-able way
    - E.g., change email, remove sensitive file
  - Check out each version, run command, commit result
- **Hooks**
  - Run scripts before commit, merging,

GitHub

- GitHub acquired by MSFT for 7.5b
- **GitHub: largest host for Git repos**
  - Git hosting (100m+ open source projects)
  - PRs, forks
  - Issue tracking
  - Code review
  - Collaboration
  - Wiki
  - Actions (CI / CD)
- **"Forking a project"**
  - Open-source communities
    - Negative connotation
    - Modify and create a competing project
  - GitHub parlance
    - Copy a project to contribute without push/write access