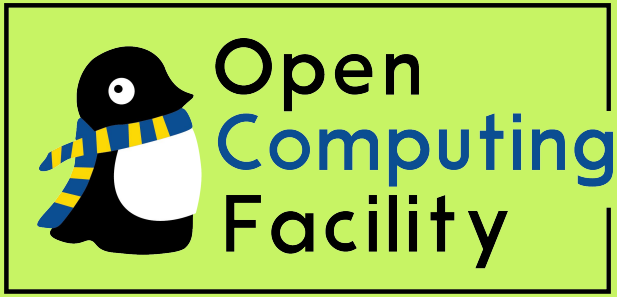


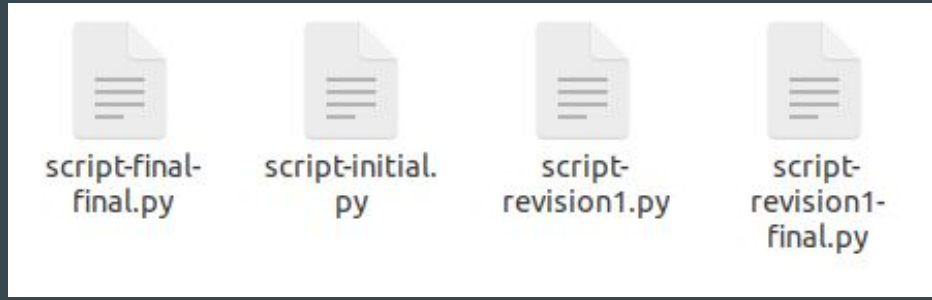
Version Control & Backups

Lecture 8

aly, trinityc

(content/slide credits Max Vogel, Hilfinger)

Why Version Control (VC)



1. Track changes to code over time
2. Collaborate with others without having to worry (too much) about conflicting changes
3. Create and test features without breaking production

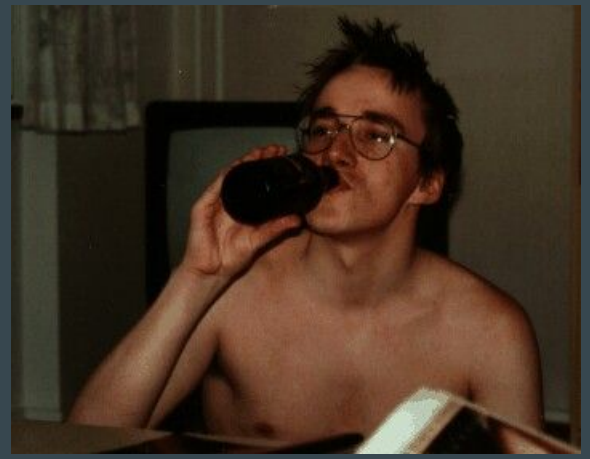
- Examples of Version Control Systems (VCS):
  - Git, Mercurial, Subversion, Perforce, Bazaar

➢ Git is the modern-day standard, we'll focus on it!

  - Git is not GitHub!

About Git

- FOSS created by Linus Torvalds in 2005 for development of the Linux kernel
  - Developer of their previous, proprietary VCS (Bitkeeper — now dead!) withdrew the free version.
  - ➢ *Torvalds has quipped about the name Git, which is British English slang meaning "unpleasant person". Torvalds said: "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."* (wiki)
  - First implementation took ~2–3 months to create



- Initially a collection of basic primitives (now called "plumbing") that could be scripted together to provide the desired functionality
- Over time, higher-level commands ("porcelain") were built on top of these to provide a convenient user interface.

What makes Git Special?

- Git stores **snapshots** (versions) of the files and directory structure of a project, keeping track of their relationships, authors, dates, and log messages.
- Git Has Integrity
  - Hashes objects with SHA1
  - Impossible to change anything without Git knowing
    - Lose information in transit
    - File corrupt
- Git is Very Fast
  - Nearly every operation is local
    - No network latency overhead
  - Browsing history of the project involves Git reading it directly from your local database
  - Works offline

Git Internals

Git represents project history as directed acyclic graph of commit nodes

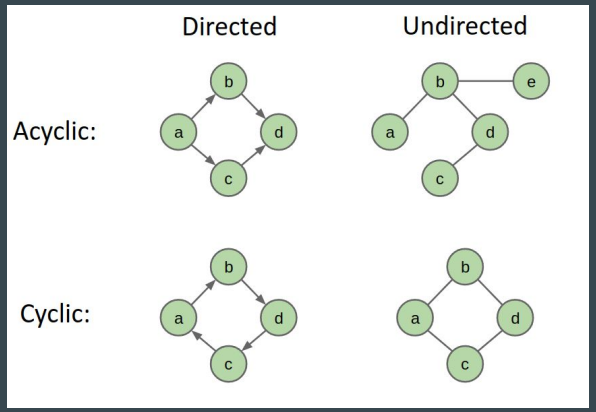
Nodes point (one-way) to the state they're *based on* and there are no cycles

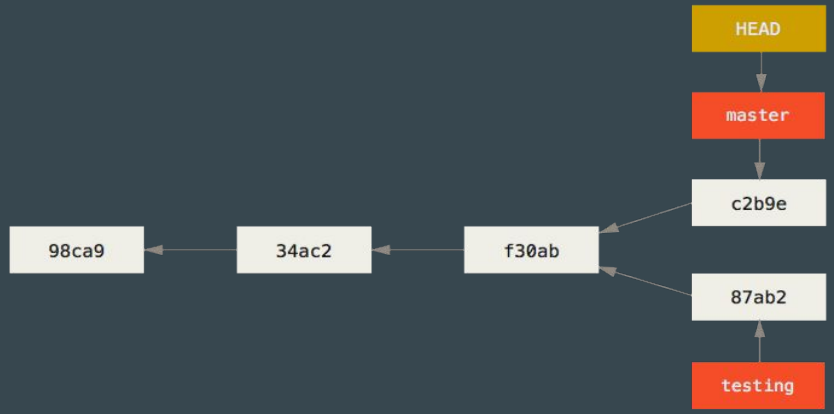
Commits correspond to project state's tree (snapshots) which are made up of:

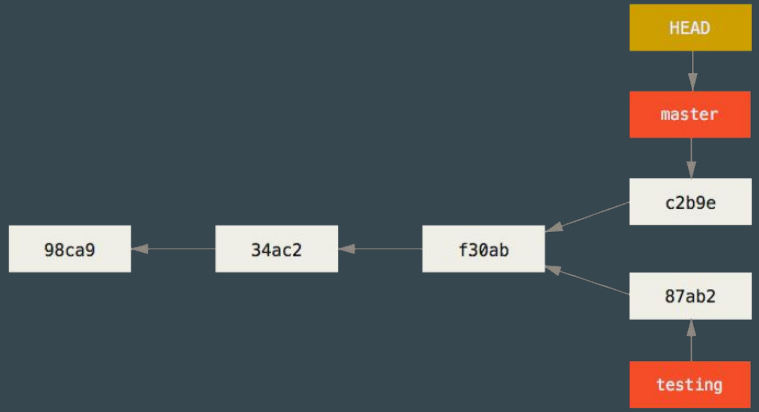
- Files: "Blobs" of bits
- Folders: "Trees" containing blobs and/or other trees

Branches are pointers to the head of a line of work. Default name is `master` (or `main`)

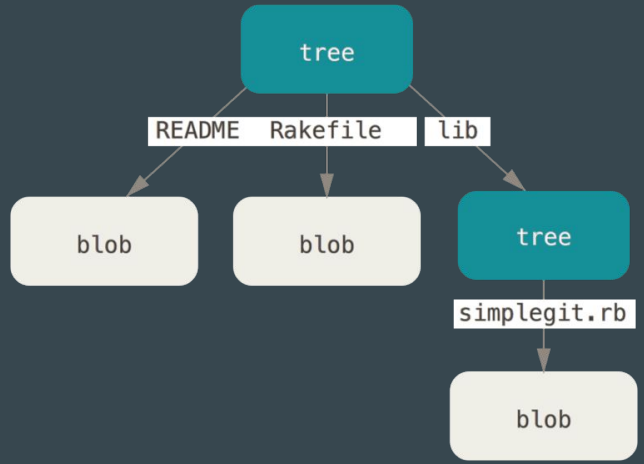
Head is a pointer to the local branch you're currently on







Distributed: There can be many copies of a given repository, each supporting independent development, with machinery to transmit and reconcile versions between repositories.

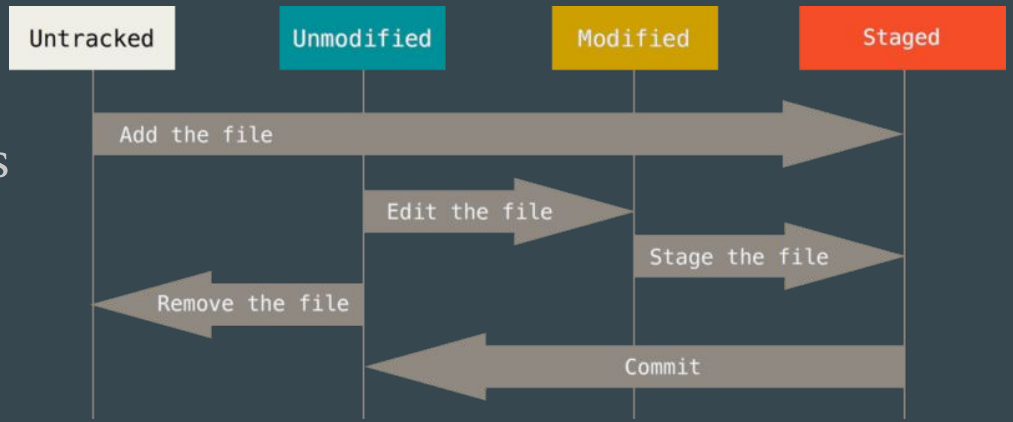


File States

- **Tracked** — files git knows about
  - Files that were in the last snapshot + newly staged (added) files
  - Tracked files are either **unmodified**, **modified**, or **staged**
- **Untracked** — everything else
  - `.gitignore` specifies intentionally untracked files to ignore

- **Modified:** File is changed but isn't committed it to your database (repository) yet.
- **Staged:** Modified file is marked in its current version to be included in the next commit snapshot.

```
$ git status
```



Getting Started: Creating a Repository

- We have an existing project in directory `proj/`

  ```
  $ cd proj/
  $ git init
  ```

  - ➢ This makes `proj/` a Git "repository"
  - ➢ Creates a new subdirectory `.git`

- We want to clone an existing repository

  ```
  $ git clone <repo URL> [destination]
  ```

  - ➢ Creates a directory **`destination`**
  - ➢ Initializes a `.git` directory inside it
  - ➢ Pulls down data for that repository from `<repo URL>`

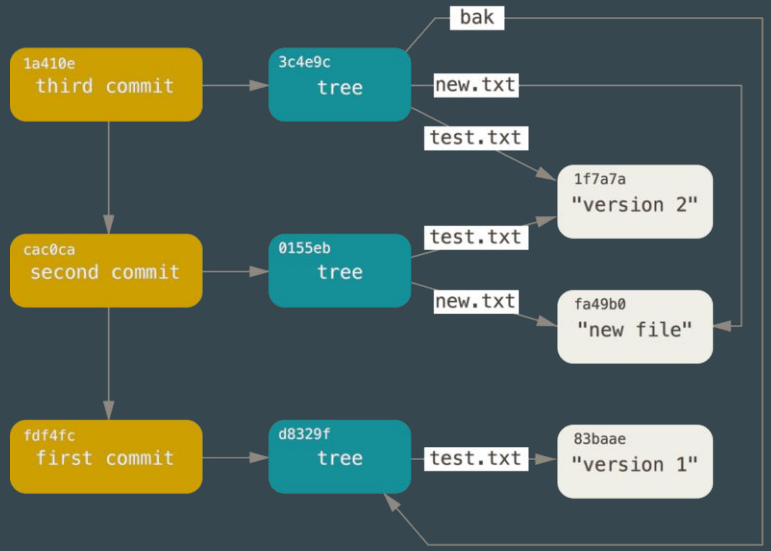
  ```
  $ git clone https://github.com/0xcf/decal-labs
  ```

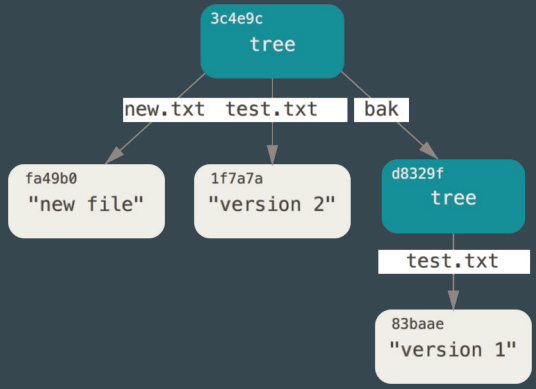
Walkthrough

```
$  git init test
$  cd test
$  echo 'version 1' > test.txt
$  git add test.txt
$  git commit -m "first commit"
$  echo 'version 2' > test.txt
$  echo 'new file' > new.txt
$  git add test.txt new.txt
$  git commit -m "second commit"
$  git read-tree --prefix=bak [first commit hash]
```

- Creates tree/directory "bak" that contains snapshot of first commit

```
$  git commit -m "third commit"
```





Use a reference!

☛ Git cheat sheet

https://education.github.com/git-cheat-sheet-education.pdf

GitHub

GIT CHEAT SHEET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history
`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker
`git config --global color.ui auto`

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit
`git add [file]`
add a file as it looks now to your next commit (stage)
`git reset [file]`
unstage a file while retaining the changes in working directory
`git diff`
diff of what is changed but not staged
`git diff --staged`
diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

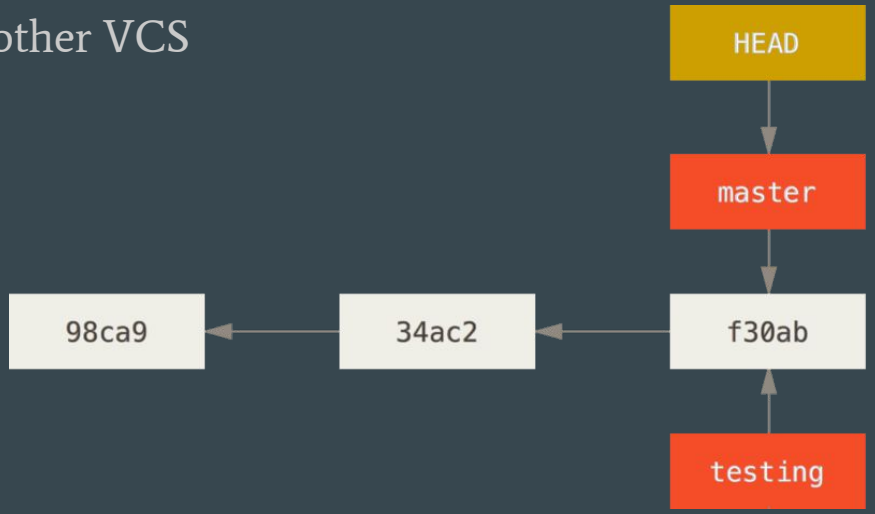
BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`

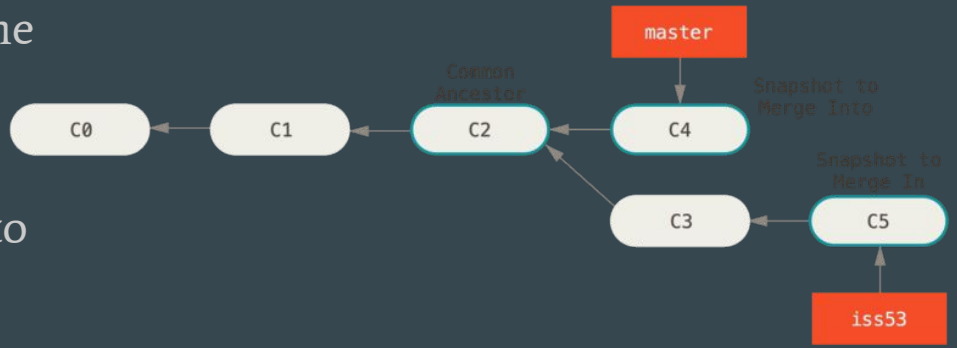
Branching

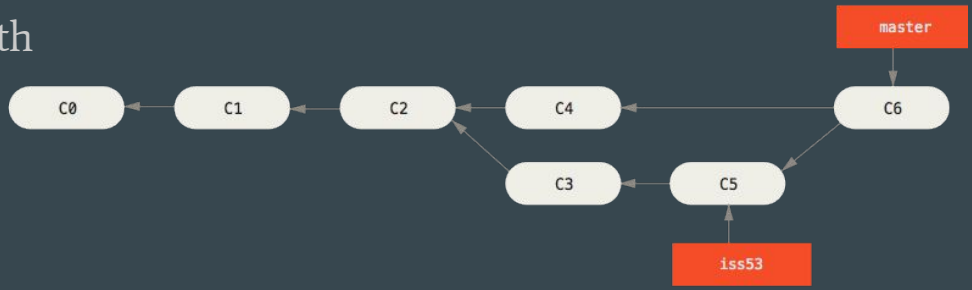
- Idea is to create a new ‘branch’ with the current branch as the ‘trunk’
  - That is, this creates a new pointer at the same location of `HEAD`
  - Every time you commit, the pointer of the active branch moves forward automatically
- Git’s ‘killer feature’ that sets it apart from other VCS



Merging

- Idea is to combine a branch back into the mainline/trunk with a merge commit
- For example, you have branch `iss53` (issue #53) and are ready to merge it into the main codebase (which has changed since you started `iss53`):
  - `$ git checkout master`
    - ➢ Change `HEAD` to point to `master`
  - `$ git merge iss53`
    - ➢ Creates a merges commit (`C6`) with `HEAD (master)` and `iss53`
  - `$ git branch -d iss53`
    - ➢ Deletes branch `iss53`





Conflicts

- If the same part of the same file is different in the two branches you're merging, Git won't be able to merge them cleanly and it will throw a merge conflict
  - Anything with conflicts will go unmerged
  - Git adds conflict-resolution markers to the files that have conflicts
- See which files conflict with `git status`
- `git add` on each file to mark it as resolved
- `git commit` to finalize the merge commit

```
<<<<<<< HEAD

[Lines of code from HEAD
(i.e master)]

=======

[Lines of code from the
rebasing/merge target (i.e
testing, iss53, dice)]

>>>>>>> [Commit message]
```

Rebase

- Idea is to take all commits from a branch and apply them on top of `HEAD`
  - No merge commit
  - As if you just made all the commits on the main branch anyways (linear history)

```
$ git checkout experiment
$ git rebase master
```

➢ Goes to common ancestor of branches (`C2`), saving the difference of each commit on `HEAD` (`experiment`) to temporary files, setting `HEAD` to the same node as master, and finally applying each change onto master.

```
$ git checkout master
$ git merge experiment
```

Remotes

- The remote/offsite copy of the repository
  - `origin` is the default name for a remote when you run `git clone`
  - Remote branch names take the form `<remote>/<branch>`

$ `git remote -v`

➢ View all remotes; their name and the URL they map to (i.e https://github.com/[...]**.git)**

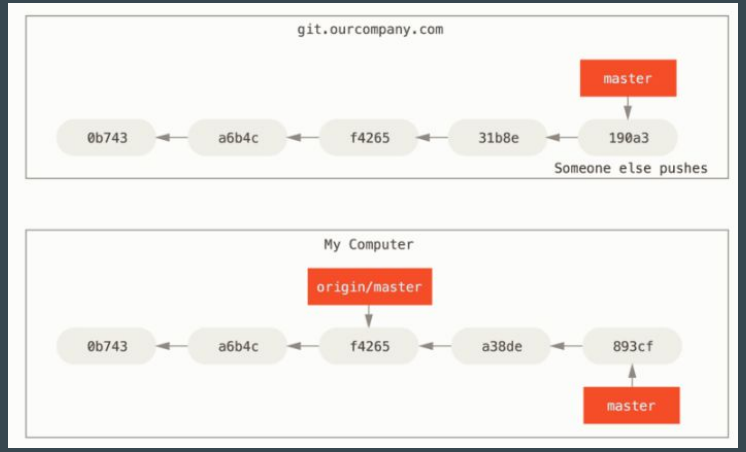
$ `git remote show <remote>`

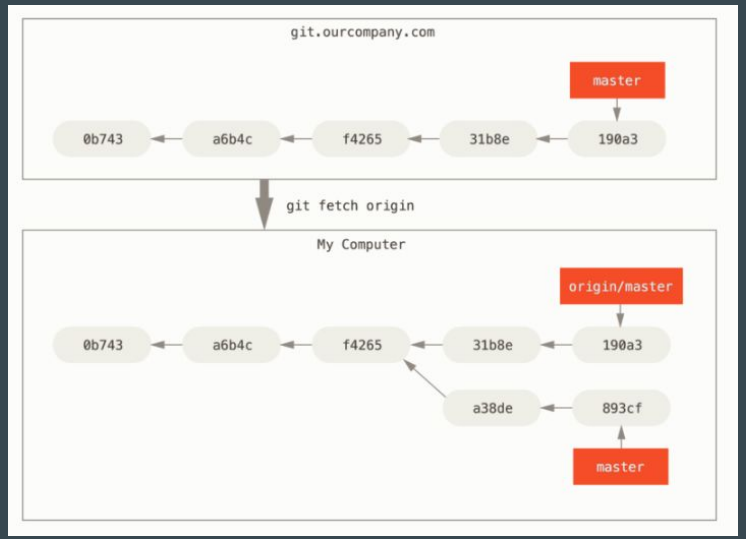
➢ Show information about `remote`

$ `git fetch <remote>`

➢ Fetches any new data from remote and updates your local database (moving the `origin/master` pointer)

➢ Will **not** modify your working directory at all





Pushing & Pulling

`$ git pull [remote/branch]`

`$ git fetch [r,b] && git merge [r,b]`

➢ Looks up what server and branch your current branch is **tracking**, fetches from that server, then attempts to merge

- When you clone a repository, it automatically creates a *master* branch that **tracks** origin/*master*
- When you checkout a remote repositories branch, it also automatically **tracks** it

- **Upstream-branch** is the remote branch that the tracked branch, well, tracks
  - If you initialize a new repository you won't have an upstream branch

`$ git remote add origin [URL]`

`$ git branch -u origin/<branch>`

`$ git push <remote> <branch>`

- Pushes your **local commits** to remote/branch
- Local branches aren't automatically synchronized; thus, you can have private branches
- Good practice to pull before you push

Summary: The Git Workflow

```
$ git checkout master
$ git checkout -b feature
    $ git branch feature && git checkout feature
```

- [Modify files]

```
$ git stage file-changed-1 [...] file-changed-n
$ git commit -m “lorem”
$ git push origin feature
$ git checkout master
$ git pull origin/master
$ git merge feature
$ git push origin master
```


BACKUPS
YOUR SHIT WILL BREAK

Backups

- Just do it
- Murphy's law
  - Accidental or malicious deletion
  - Device failure
  - Software failure
  - And a Berkeley special, theft.
- Automated backups because you will forget
- Don't leak information! Backups must be secure
- Make sure your backups actually work!
  - Routinely test backup procedure and recovering from backups

The 3-2-1 Rule

3. Have at least 3 copies of your data

2. Store your data on at least 2 different media
    E.g. 1 hard drive, 1 backup server/computer

1. Have at least 1 copy of your data off-site
    E.g. on Amazon S3, “the cloud,” under a mattress

What happens if you don't follow what I said

- GitLab 1/31/17 Database Outage
- Engineer accidentally runs `rm -rf` on their production PostgreSQL database
  - Noticed after 1 second and CTRL-C'd, but lost 300GB of production data already
- This shouldn't have been too bad - 300GB isn't THAT much data nowadays, and they can just recover from a backup, right?

Backup 1: Amazon S3

- GitLab had an automated process to upload a backup to Amazon S3 (Amazon file storage) every 24 hours
- GitLab engineers inspected their S3 bucket, hoping to find a backup
- Turns out their backups had been failing for weeks due to a version mismatch, and their notification system was broken too

Backup 2: Azure Disk Snapshots

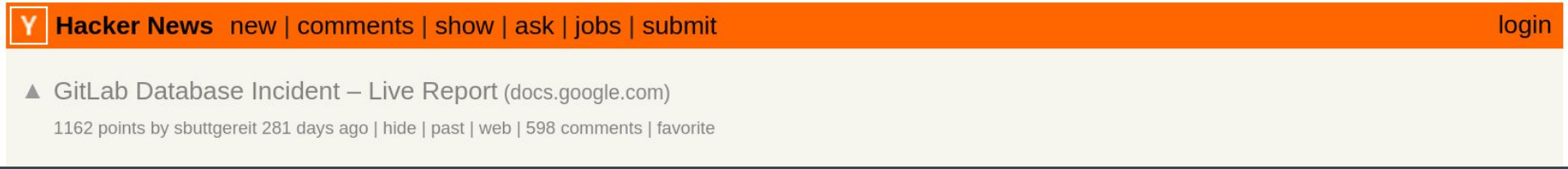
- GitLab runs on Azure (Microsoft Cloud Hosting Provider)
- Azure offers the option to generate snapshots of an entire disk periodically
- GitLab had enabled this to run every 24 hours...
- ....except on the database servers, because they thought they had enough backups

Why GitLab still exists today: Hail Mary LVM Snapshots

- LVM: Logical Volume Manager
- Not meant to be a backup, but luckily they had these
- Every 24 hours, copy data from prod to staging environment
- An engineer had run this ~6 hours before the incident luckily enough
- Unfortunately, took GitLab **18 hours** to recover, since staging was not meant for data recovery process
  - Different region and slow disks

Impact

- “It's hard to estimate how much data has been lost exactly, but we estimate we have lost at least 5000 projects, 5000 comments, and roughly 700 users.”
- Became both a feel-good story of transparency and not firing the engineer involved but also a WTF story about their backups on HackerNews
- Good lesson on the importance of keeping backups, making sure they work, and practicing recovering from them



Tools for Backups

- rsync
  - Simple command-line util for local <-> remote transfer
  - Skips copying files that are the same @ destination, so good for backups
  - Uses SSH for transferring to remote hosts

```
$ rsync -av -P [source] user@host:[destination]
```

- Rclone: “rsync for cloud storage”
  - Supports every major cloud storage provider

```
$ rclone sync source:path dest:path
```

  - Can mount cloud storage as local FS

More

- Pro Git — git-scm.com
  - The bible
  - Ch 1–5 form good foundation
  - Source for all the tree diagrams
- Oh Shit, Git!?! — ohshitgit.com
  - ➢ *“Git is hard: screwing up is easy, and figuring out how to fix your mistakes is fucking impossible”*
  - Short guide on how to recover from some common Git mistakes
- Shell/Editor integration
  - Vim-fugitive