Version Control

Based on Aaron's

What is “Version Control”

- A system that records changes to a file or set of files over time so that you can recall specific versions later.
- Keep every version of an update for a project
- Able to
  - restore **selected files** back to a previous state,
  - restore **the entire project** back to a previous state,
  - compare changes over time,
    - see who last modified something that might be causing a problem,
    - who introduced an issue and when, and more.

Based on Aaron’s

Local Version Control Systems

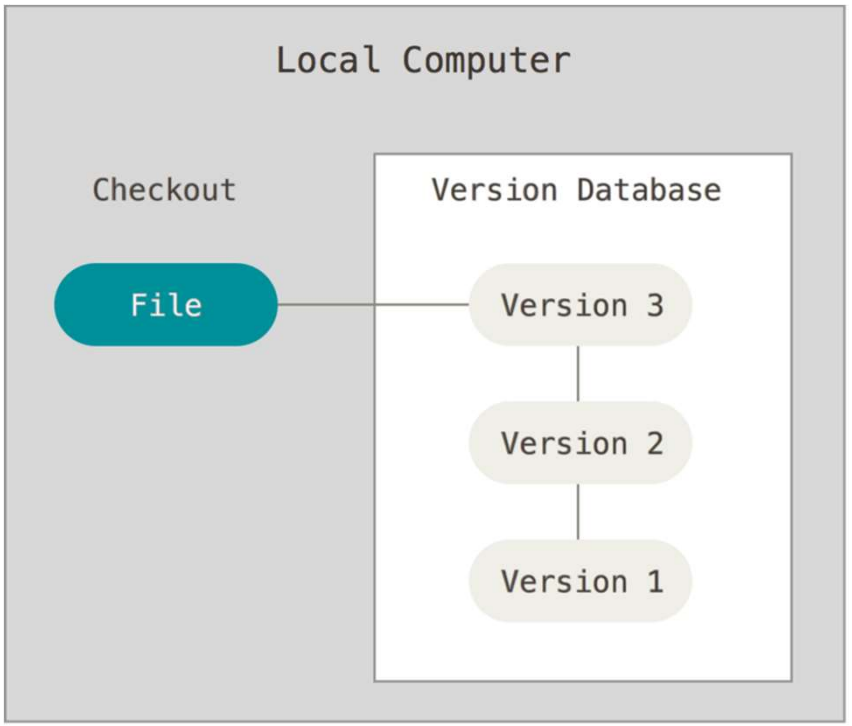
- The simplest version-control method of choice is to copy files into another directory.
- Very simple and common approach, but it is also error prone.
  - Easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.
- To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.

Based on Aaron's

Local Version Control Systems

https://en.wikipedia.org/wiki/Revision_Control_System

- One of the most popular VCS tools was a system called Revision Control System, RCS.
- RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.



Based on Aaron's

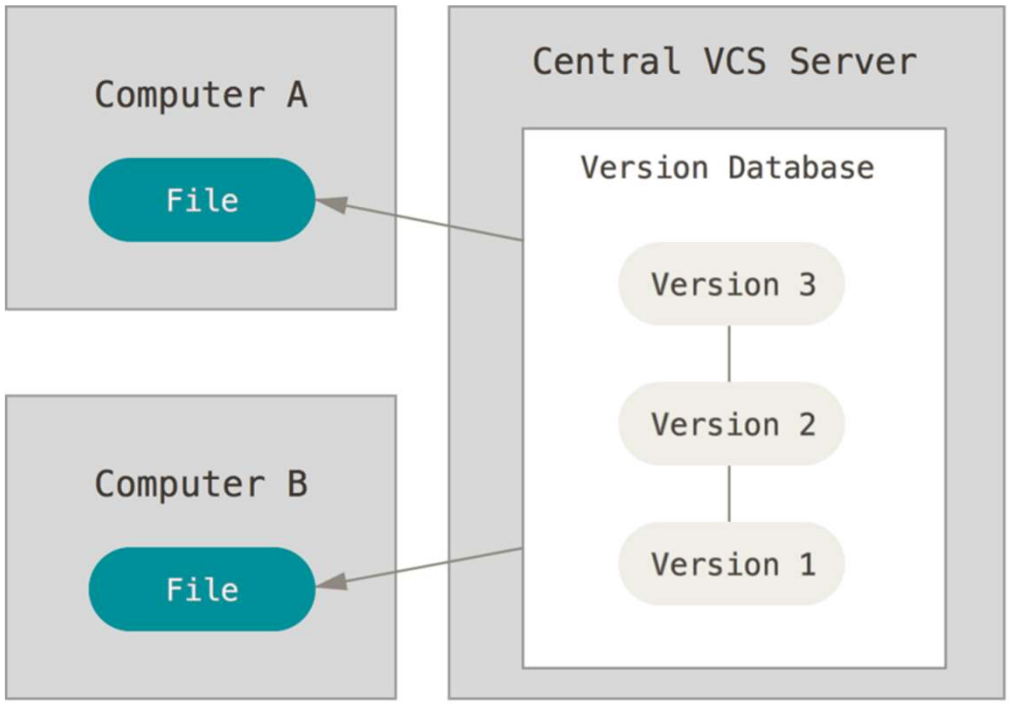
Centralized Version Control Systems (CVCS)

- The next major issue that people encounter is that they need to collaborate with developers on other systems.
- To deal with this problem, Centralized Version Control Systems (CVCSs) were developed.
- These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place.
- For many years, this has been the standard for version control.

Based on Aaron's

Centralized Version Control Systems (CVCS)

- This setup offers many advantages, especially over local VCSs.
  - Everyone knows to a certain degree what everyone else on the project is doing.
  - Administrators have fine-grained control over who can do what.
  - Easier to administer a CVCS than it is to deal with local databases on every client.



Based on Aaron's

Centralized Version Control Systems (CVCS)

- Serious downside: Single Point of Failure
  - If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.
  - If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything—the entire history of the project except whatever single snapshots people happen to have on their local machines.
    - Local VCS systems suffer from this same problem—whenever you have the entire history of the project in a single place, you risk losing everything.

Based on Aaron's

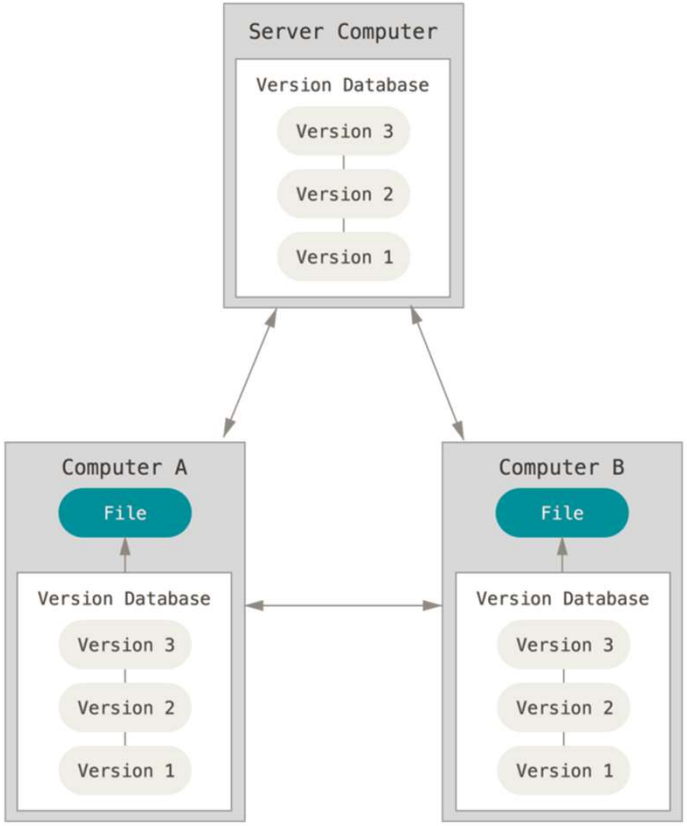
Distributed Version Control Systems (DVCS)

- In a DVCS (such as Git, Mercurial),
  - fully mirror the repository, including its full history.
  - clients don't just check out the latest snapshot of the files
- If any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it.
- Every clone is really a full backup of all the data

Based on Aaron's

Distributed Version Control Systems (DVCS)

- Many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project.



Based on Aaron's

History of GIT

Based on Aaron's

SIM GLOBAL EDUCATION
A Member of the SIM Group

Source Code Control System:

- written for the IBM system 370.
- ported to early versions of Unix.

It works as a utility locking:

- requesting by way of a queue
- updating code and files.
- SCCS uses versions, kept in a special file system directory called a repository.
- A user checks out a file, and the file is locked so others can't use it.
- After a user updates it and checked out, the file is committed back into the repository
- the versioning information is updated.

SCCS was designed to resemble the physical world of punch cards, where a programmer could physically take the code, modify it and return it to a library

Based on Aaron's

Concurrent Version Systems (CVS)

- In the 1990s, Concurrent Version Systems (CVS) was the most popular version control system.

- CVS allowed large teams to work distributively, and
- It focused on merging more than it did on tracking.
- Dr. Dick Gruento developed it for his students around the world, to collaborate on the Amsterdam compiler kit, which began in 1986.

Based on Aaron's

Concurrent Version Systems (CVS)

- CVS implements a client-server architecture
- It allows code-bases to branch and be merged later.
- Merging process can be done automatically or manually.
- CVS allows more than one person to work on a file at the same time.
- It just requires the same people who are changing the files if any conflicts arise

Based on Aaron's

Subversion (Early 2000)

- It can track entire projects
- It implements releases which means a code-base can be in a state ready to be published.
- It solves the atomic commit problem, that is, when two users commit a change to the same line, it expects the user who committed the most recent change to solve that conflict.
- Subversion also had better support for directories and non-source code files.
- It works on Windows, Linux, and various programming IDs.
- It also supports distributed and replicated servers, allowing for higher performance.
- It also does automated backups of code repositories

Based on Aaron's

GIT

- It was written by Linus Torvalds,
- The purpose: to fix Subversion (early GIT)
- Since then Git his largely been expanded. Git introduces the idea of a pull request.

Based on Aaron's

GIT

- Currently, Git and Mercurial are the most popular VCS.

- Both Git and Mercurial focus on history and branching, not just tracking.
- The history allows users to implement and fix past bugs and understand the design process for a large project.
- They also implement branch semantics. For example, cloning a repository means that you've retrieved a repository from a remote location, and you're going to allow changes that you make to be committed back to that repository.
- Forking a repository, means that you've copied one remote repository into another remote repository, in which you might commit changes back to.
- Merging combines branches or repositories with one another.
- Both Git and Mercurial retain the typical update and commit routines apparent in all version control systems.

Based on Aaron's

Git

- Some of the goals of the new system were as follows:
  - Speed
  - Simple design
  - Strong support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like the Linux kernel efficiently (speed and data size)
- Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's amazingly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development

Based on Aaron's

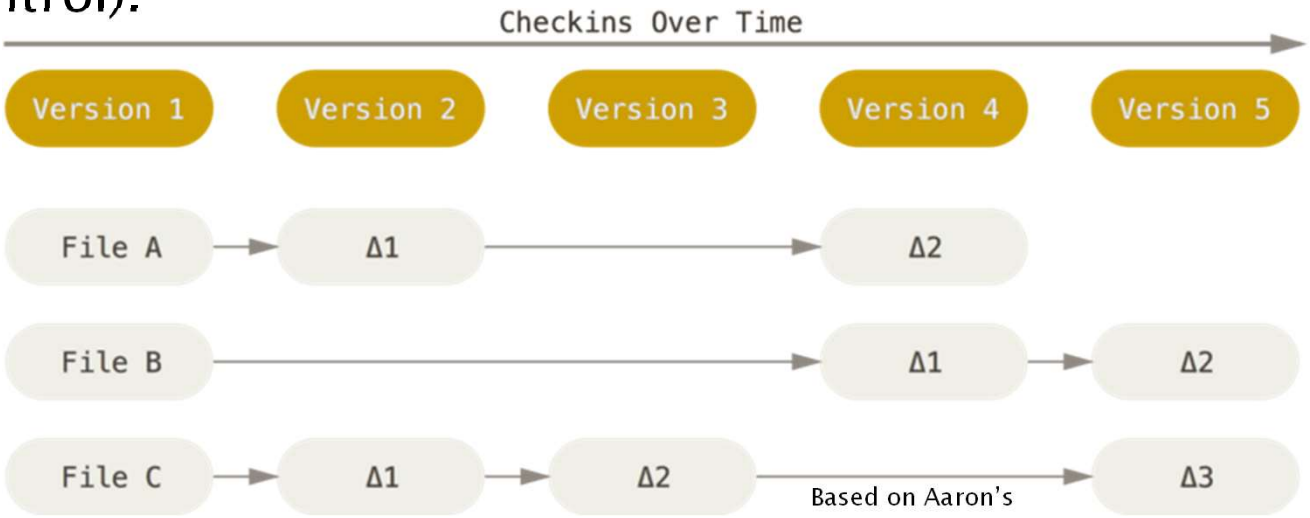
Snapshots, Not Differences

- The major difference between Git and any other VCS (e.g. Subversion) is the way Git thinks about its data.
- Conceptually, most other systems store information as a list of file-based changes (incremental).
  - information they store as a set of files and the changes made to each file over time (this is commonly described as delta-based version control).

Incremental means just to load the new records available after the previous load

If the last # of records were 100 and today 20 more records. We will be loading only latest 20 records in today's job run.

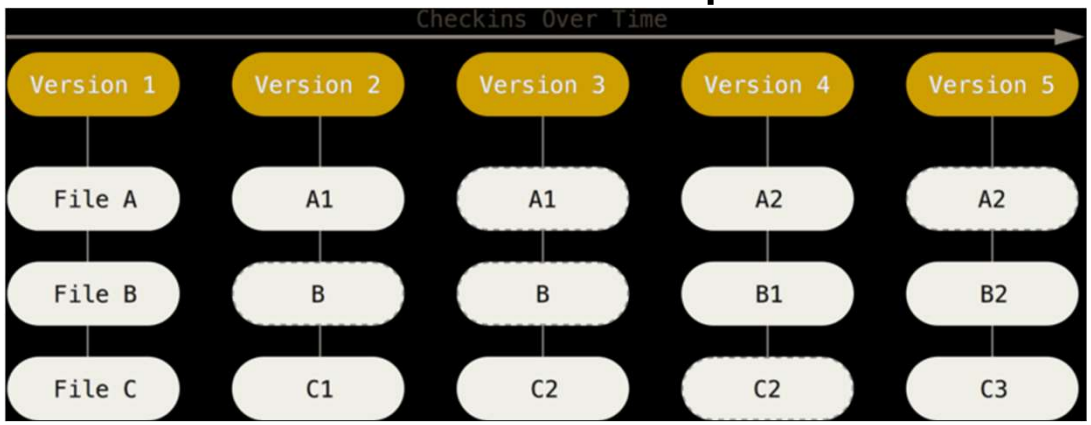
SNAPSHOT of data your concern is complete data available.



MoRE: https://courses.csail.mit.edu/6.S194/13/lessons/03-git/vcs.html

Snapshots, Not Differences

- Git thinks of its data more like a series of snapshots of a miniature filesystem.
- Every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
  - To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.
  - Git thinks about its data more like a stream of snapshots.



Based on Aaron's

Nearly Every Operation Is Local

- Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network.
  - Compare to CVCS where most operations have that network latency overhead,
- Because the entire history of the project right there on your local disk, most operations seem almost instantaneous.

Based on Aaron's

Nearly Every Operation Is Local

- For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you—it simply reads it directly from your local database. This means you see the project history almost instantly.
- If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

Based on Aaron's

Nearly Every Operation Is Local

- This also means that there is very little you can't do if you're offline or off VPN.
  - If you get on an airplane or a train and want to do a little work, you can commit happily (to your **local** copy) until you get to a network connection to upload (push).
- In many other systems, doing so is either impossible or painful.
  - In Perforce, for example, you can't do much when you aren't connected to the server;
  - in Subversion and CVS, you can edit files, but you can't commit changes to your database (because your database is offline).
- This may not seem like a huge deal, but you may be surprised what a big difference it can make.

Based on Aaron's

Git Has Integrity

- Everything in Git is checksummed before it is stored and is then referred to by that checksum.
  - This means it's impossible to change the contents of any file or directory without Git knowing about it.
- The mechanism that Git uses for this checksumming is called a SHA-1 hash.
  - A 40-character string composed of hexadecimal characters (0-9 and a-f) and calculated based on the contents of a file or directory structure in Git.
  - A SHA-1 hash looks something like this:
    - 24b9da6552252987aa493b52f8696cd6d3b00373

Based on Aaron's

The Three States

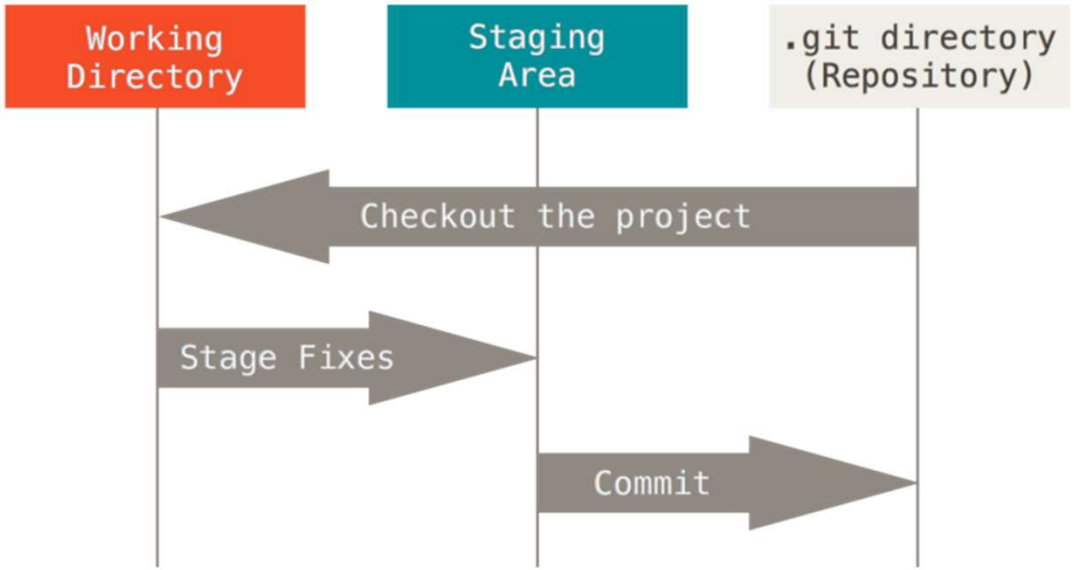
- Three main states that files can reside in: **modified**, **staged**, and **committed**:
  - Modified
    - Have changed the file but have not committed it to the database yet.
  - Staged
    - Have marked a modified file in its current version to go into your next commit snapshot.
  - Committed
    - Data is safely stored in local database.

Based on Aaron's

The Three States

- The working tree
  - A single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- The staging area
  - A file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.
- The Git directory
  - Where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you **clone** a repository from another computer.

- This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.



Based on Aaron’s