Contents

[Getting Started 2](#_Toc56085779)

[About Version Control 2](#_Toc56085780)

[Local Version Control Systems 2](#_Toc56085781)

[Centralized Version Control Systems 2](#_Toc56085782)

[Distributed Version Control Systems 3](#_Toc56085783)

[Snapshots, Not Differences 3](#_Toc56085784)

[The three states 4](#_Toc56085785)

[Git Basics 5](#_Toc56085786)

[Working with Remotes 5](#_Toc56085787)

[Tagging 5](#_Toc56085788)

[Annotated Tags 6](#_Toc56085789)

[Lightweight Tags 6](#_Toc56085790)

[Sharing Tags 6](#_Toc56085791)

[Deleting Tags 6](#_Toc56085792)

[Checking Out Tags 6](#_Toc56085793)

[Git Aliases 6](#_Toc56085794)

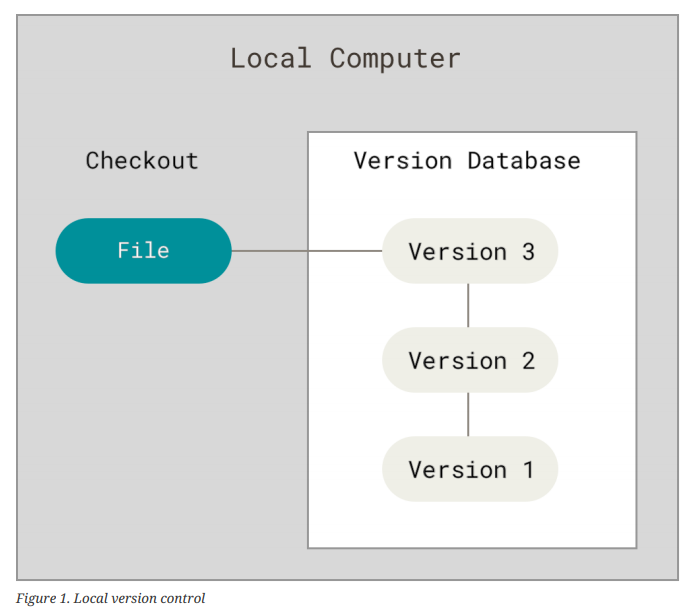
Git Architecture

# Getting Started

## About Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

## Local Version Control Systems

RCS or Revision Control System

## Centralized Version Control Systems

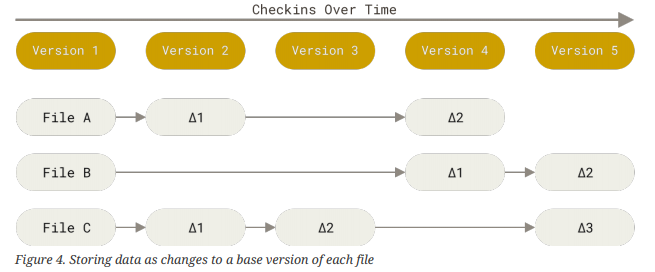
However, this setup also has some serious downsides.

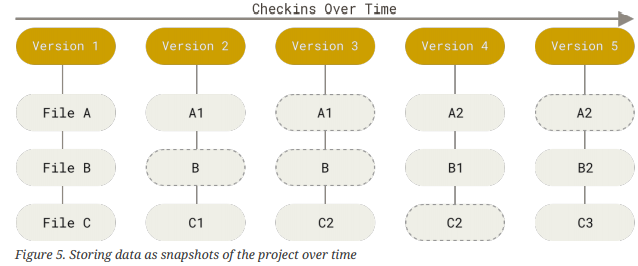
* The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on.
* If the hard disk the central database is on becomes corrupted, 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.

## Distributed Version Control Systems

### Snapshots, Not Differences

The major differences between Git and any other VCS is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes.

These other systems think of the information they store as a set of files and the changes made to each file over time. (delta-based version control)

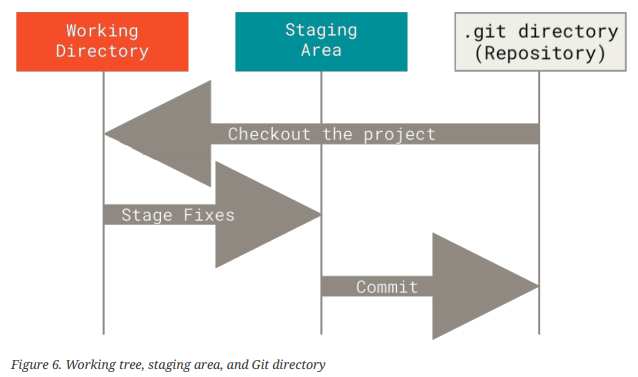
Git doesn’t think of or store its data this way. Instead, Git thinks of its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. (stream of snapshots)

### The three states

The three main states that your files can reside in:

* *Modified* - that you have changed the file but have not committed it to your database yet.

The working tree is 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.

* *Staged* - means that you have marked a modified file in its current version to go into your next commit snapshot.  
  The staging area is 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.
* *Committed* - means that the data is safely stored in your local database

# Git Basics

git rm <filename>

git log –stat  
git log –pretty=oneline  
git log –since=2.weeks

git diff 🡺 That command compares what is in your working directory with what is in your staging area.

git diff –staged 🡺 This command compares your staged changes to your last commit.

git commit  
git commit –amend

git reset HEAD <file>  
git reset HEAD –hard

git restore –staged <file>

Note: Any local changes you made to that file are gone — Git just replaced that file with the most recently-committed version. Don’t ever use this command unless you absolutely know that you don’t want those unsaved local changes.

## Working with Remotes

git remote -v  
git remote add <short-name> <URL>

git fetch <remote>

git pull

Note: From git version 2.27 onward, git pull will give a warning if the pull.rebase variable is not set. Git will keep warning you until you set the variable.

If you want the default behaviour of git (fast-forward if possible, else create a merge commit): git config --global pull.rebase "false" If you want to rebase when pulling: git config --global pull.rebase "true"

git push origin master

git remote show origin

git remote rename <old-name> <new-name>

git remote remove <remote>

## Tagging

Like most VCSs, Git has the ability to tag specific points in a repository’s history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on).

git tag  
git tag -l <tag-name>

Git supports two types of tags: *lightweight* and *annotated*.

A lightweight tag is very much like a branch that doesn’t change – it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They are checksummed; contain the tagger details; have a tagging message and can be signed and verified with GNU Privacy Guard (GPG).

### Annotated Tags

git tag -a v1.4 -m “my version 1.4”

git show v1.4

### Lightweight Tags

git tag v1.4-lw

git show v1.4-lw

git tag -a <tag-name> <commit-name>

### Sharing Tags

git push origin <tag-name>

### Deleting Tags

git tag -d v1.4-lw

Note that this does not remove the tag from any remote servers. There are two common variations for deleting a tag from a remote server. The first variation is **git push <remote> :refs/tags/<tag-name>**

The second (and more intuitive) way to delete a remote tag is with:

**git push origin --delete <tag-name>**

### Checking Out Tags

git checkout <tag-name>

## Git Aliases

$ git config --global alias.co checkout

$ git config --global alias.br branch

$ git config --global alias.ci commit

$ git config --global alias.st status

$ git config --global alias.unstage 'reset HEAD --'

$ git unstage fileA

$ git reset HEAD – fileA

$ git config --global alias.visual '!gitk'