# Version Control and Git Statistics 506

## Version Control

Version control is a structured way of managing changes to files (whether they be written documents, code, images, or anything else). We've all used the most basic form of version control - when you make a change to a document, you save it. This has an obvious limitation - if you want to undo a change (not including just hitting "Undo" for the most recent change(s) while modifying the file), you're out of luck.

Some of you may have developed your own structure for a more rigid version control: either manually or automatically creating backups of files, capturing a snapshot of the current state of the file. This is a massive improvement over no older versions, but still has serious limitations, such as the need to keep many versions of a file around and hope you don't accidentally open an older one.

When people refer to Version Control, they usually mean a more deliberate variation with a more rigorous systems for storing these file changes. These days, the most common tool for Version Control is **git**.

### Other Version Control tools

There are many other version controls tools, but most are extremely niche. The only other two worth mentioning are subversion, usually called "svn", and mercurial. These were the dominant Version Control tools prior to git. Some open-source software still relies on them, but it's getting more and more rare. There are plenty of resources online explaining the pros and cons to each system if you're interested.

## git vs GitHub

When most people hear of "git", they probably think of "GitHub" (if anything). GitHub is a website that hosts git **repositories**, but git is agnostic to GitHub - you could host your repositories on any server. You can also use git entirely locally without any online component, though that loses one of the primary benefits of a version control system, off-site backups.

We will be using GitHub in this class, and will discuss the GitHub client below. This first section is about git more generally.

### git

A git **repository** holds the files for a given project. In addition, it contains a list of all changes to the files. Whenever you're happy with the changes you've made to your files, you make a **commit** which updates the files in the repository to the version you're currently working with, as well as saving the commit as a **diff**, that is, a list of the changes.

For example, imagine you had this shopping list in your repository:

```
- apples
- ketchup
- white bread
- eggs
```

You need to modify this list, so you update it to the following:

```
- apples
- ketchup
- whole wheat bread
- eggs
- lemons
```

You now commit this change to the repository. The following two things happen:

1. The repository now stores the updated lists.
2. The repository stores a diff tracking that `- lemons` was added at the end of the file, and that `white` was removed and `whole wheat` added on line 4.

This diff is called a "commit".

Note that the previous version of the file is *not saved*. This makes git much more efficient - if you had a 10,000 line file and change one word, you don't need to save two copies of that 10,000 files, you save one copy and a 1-line diff. You can restore any previous versions of the file by asking git to replay the changes up to that point.

#### Commit messages

When making a commit, you must provide a message describing the changes you've made. These messages should be concise but descriptive. Poorly written messages make it very difficult to browse the history of the commits.

## Local versus remote repositories

Git is a decentralized system. There can be many copies of a repository and they can pass commits amongst themselves such that none is the canonical source. Users often declare a single repository (most commonly, the one hosted on GitHub) as the canonical version, but the git software does not enforce this.

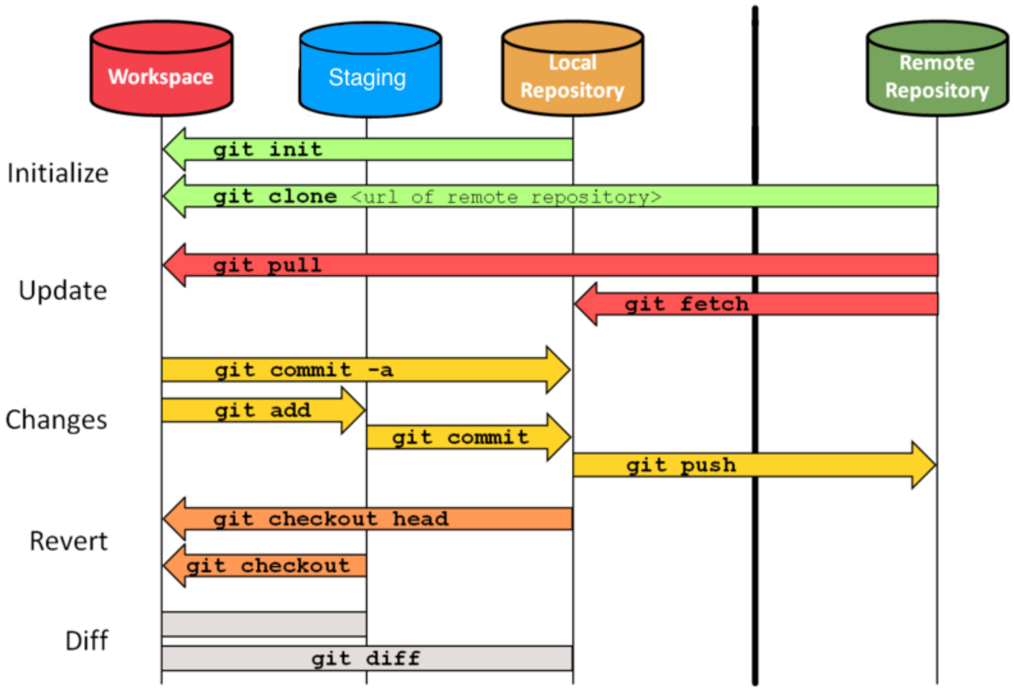
The typical workflow with an existing repository that you've not interacted with would look like:

1. **Clone** a remote repository to your local machine so that you can work on it.
2. Modify the files as desired.
3. **Add** changes to staging - often you may want to commit all changes, but a lot of the time you only want to commit some changes right now and keep working on other.
4. **Commit** the changes to your local repository.
5. **Push** the new changes from your local repository to the remote repository.

This workflow would like very slightly different if you are returning to work on a repository you've worked on in the past:

1. **Pull** from the remote repository to your local repository to make sure your local repository is up-to-date.
2. Modify
3. **Add**
4. **Commit**
5. **Push**

Here is a visualization of moving between the various states:



(Source: https://sselab.de/lab2/public/wiki/sselab/index.php?title=Git but there are many other similar images online).

This includes several moves which are not discussed.

Some people recommend using **Fetch** and **Merge** instead of **Pull**, e.g. https://longair.net/blog/2009/04/16/git-fetch-and-merge/comment-page-2/

## Managing conflicts

The real magic in git comes when a file gets modified differently on two different copies of the repository. Imagine you have some remote repository, and you create a local clone. You commit some changes, but **do not** push it back to the remote.

Next, you go to another computer, clone the remote repository (which doesn't include the commits you made locally on the first computer), make different changes, commit and push.

Finally, you return to the first computer and push the changes you'd committed a while ago.

What's going to happen? The remote repository now differs from your local repository. Git will first try and handle this automatically - if the changes on the two computers are on different files, or at least different locations of the same file, it will just automatically **merge** these two commits and the remote repository will contain all the changes.

On the other hand, if you are changing the same part of the same file, it will stop the merge, and require you to fix it.

This comes in extremely handy when collaborating with other people. If you're all working on the same file, you can make your changes without worrying if others are making changes at the same time.

## git software: GUI vs command line

Git is a command line tool - it does not require using a graphical user interface. However, many new users will feel more comfortable using a GUI. GitHub Desktop (discussed below) is popular, and there are third-party clients like GitKraken.

In this class, we will primarily be interacting with git through RStudio's interface.

## Advanced topics

Git can do many more complex things. Here's a non-exhaustive list of a few things that you can look up online if you want to look into.

### Branches

If you think of the list of all your commits as series of nodes with arrows between them (a DAG), you could imagine the commits splitting into two paths. These paths are called **branches** and they allow you to modify your files and commit them without disturbing the primary code. Each branch has a name; the primary code is called the "main" branch. It was previously called the "master" branch, so you may see either if you look at online resources or at older git repositories.

### Rebasing

Say you want to remove a commit you made a while ago, *without removing all the commits between then and now.* Rebasing is a way of removing a single commit and then re-applying all other commits in order.

### Blame

In git parlance, "blame" for a line of code refers to the last commit which modified it. This can be useful for seeing the context in which the code as modified.

## Additional resources

Read more about version control and git here. In particular, you could read:

- What is version control?
- What is git?
- All the material under Getting Started.