Introduction

Ever felt like comparing how your code is now compared to a month ago? Have you found some code that you would like to 'undo'?

For my first project, my friends and I would communicate using email. We sent each other code in a zip file, with all the relevant libraries wrapped together. Soon, we realised how tedious this was.

My friends would change some code in Person class, and I would simultaneously make changes on the Account class. After downloading each other's code, we would have to copy and paste the changes.

Some changes would create compilation errors. Even worse, not all changes were isolated. We often had arguments over whose code should be accepted. All of these synchronization problems for a team size of 2!

It was during my software engineering training that I got to know about version control. It's simply storing history for your code, so that you can revert changes or compare between different versions.

There are many version control systems. Google Docs is a good idea. But the system which is designed for managing code, and hence is the most popular among software engineers, is Git.

This series will talk about how Git was made and why it works. At the end of this series, you will have a deep understanding of Git, and can create your own version control system if you like! See you in the next video!

Part 1 - What is Git fundamentally?

What is Git fundamentally? To end users, it's a version control system. But to it's designers, it is a *content addressable system*.

Let's break those words down.

1) *Content* stands for data to be stored in git. All data in git is represented as objects.
2) *Addressable* means the content can be accessed using a key or an address. You store some data, and Git returns you an ID. With this unique ID, you can ask Git to retrieve the data as and when required.
3) *System* We need a method in the madness. A system defines how the IDs will be generated, how the data will be stored, and how the interface will interact with the user.

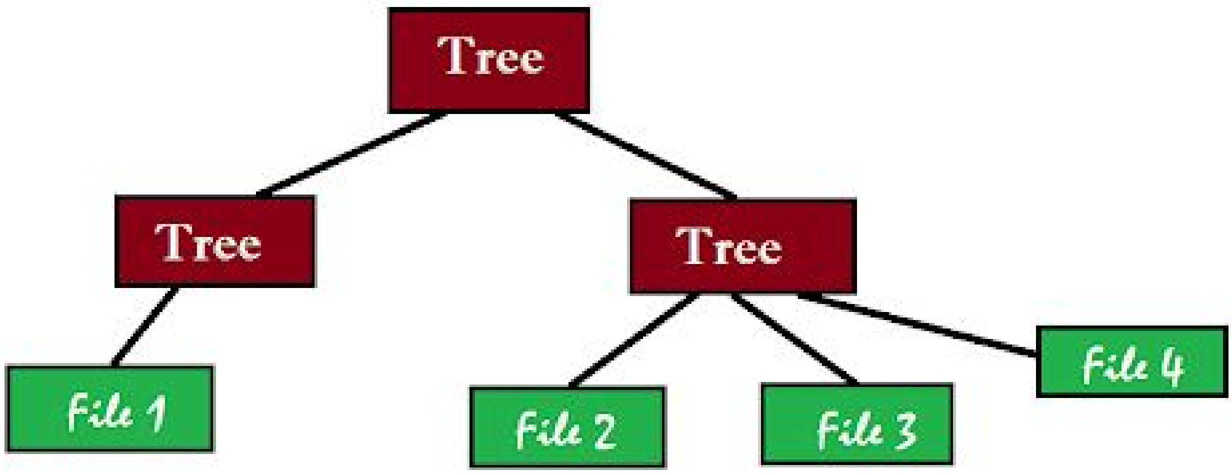
Commands:

```
mkdir MyProject - create a new folder
git init MyProject - tell git that this project will need version control
cd MyProject
ls -la (You can see the .git folder)
vim file.txt (Write some stuff - I am learning Git today)
git add file.txt
du -c (look at the new object created)
echo -e 'blob 14\0Hello, World!' | shasum
```

What does this big filename mean?
Show how data is stored using zlib for compression. (append to end does not change in zlib)
Unzip and show data in file.
Specify attributes of type, size and data which are stored in the file.
Use git pretty print to do the same.
Try to hash string and get different string from that in git.
Show format "null character" and data.
Now the hash is the same as file name.
Mention file bucketing as hash bucketing.
Show diagrams to hammer the point home.

Part 2 - File Structure

How do you rename a file in Git? If the file does not store metadata like filename and path, we can't tell the structure of the project. It must be stored elsewhere.



The metadata of files are stored in trees. These are like unix directories, containing the names of each file it stores and the ID of the file. If a file is renamed or moved, the contents of the file do not change. This means only tree objects need to be changed, which makes file renaming and moves efficient in Git.

Commands:

```
git commit -am "My first file is added"
cd .git
du -c
Pretty print the tree object
```

Part 3 - Commits in git

How do we save a snapshot in Git? As the designers, we would need to store every file in the repository. That information is contained in the root folder, which is a Tree object in Git. We can use this folder to recursively build the whole snapshot using the IDs stored in the root tree.

The snapshot should store some other information too, like who is the author of this save. The timestamp of the save, etc…
In git, a save or snapshot is called commit.

1) Make a commit using commit-tree.
2) Make a parent of that commit.
3) Show them the git log.
4) Explain contents of the commit object.
5) Show the commit structure is a graph, with merges and branching.

Great, you now have a fully functional git repo, using just the plumbing commands!

Part 4 - Compression in Git

Make a minor change to an image file. This creates a new file in Git after commiting.
This results in small changes taking very long to transfer across the network, and eventually eating a lot of disk space as well.
git gc --aggressive
Display contents of pack file

1) Pack files contain compressed git objects, with some having parent objects
2) The parent objects are similar to their children
3) Show parent child relationship by git cat parentFile and git cat childFile

This saves bandwidth and space.
Git garbage collection runs when transferring over the network. That's called pulling or pushing in Git. It also runs periodically to save disk space.
An index is used to make reading the pack file efficient. (use the verify-pack command to display the file contents)

Part 5 - Branches

We need a way to create a copy of the project to work with. The snapshots saved will be our own. This way, we separate work done by different team mates to a single workflow.
This is done using branches in Git. A member can open a branch having relevant commits to a feature.
What does a branch point to? It points to a commit.

Git checkout -b my_feature
Show the branch info stored in .git/refs/heads/my_feature
Git changes the commit pointed to by the working branch on each commit. In this way, we can work on different branches and merge them as a team.

Part 6 - Rebasing

Let's start with the important questions.

1) What is rebasing?
2) Why should we learn it?
3) How many people have lost their lives due to a bad rebase?

Rebasing is a way in which we take a series of commits and change their base commit. This is very useful if you'd like the history to be clean.

Let's say you want to add a feature to your project. The commits are related to optimizing the database connections. When you are done, you notice that the master has been changed by your teammate, who was working on the network calls.

You could merge with master, to get a graph like this. But instead, if everyone uses a rebase to merge with master, we get all merge commits, and make a single line of coherent changes.

This makes the graph of commits easier to reason about. As the changes have lesser sources and a clear progress, the team can work more efficiently. Of course, all of this is subjective and dependent on us looking through the git commit history to understand how this code was developed. Which may not be the case for small or proof of concept projects.

1) Make a feature branch. Show the git log.
2) Rebase on master. Git log has commits which are different from the original commits. The commits are also sequential without any merge commit.

The rebase essentially changed the commit history by making it appear as if someone had pulled off the new master. That is what rebase is at its core. A way to manipulate our commit history. Let's look at rebase interactive.

You can see many options here. Squash is a useful one. Say there is a set of small changes, like changing properties files, and you don't want multiple commits for them. Using squash, you can condense these commits to a single one.

Similarly, you can edit the commit message, discard a commit, etc…
The important thing to note is that rebase is a way of changing history to a more coherent graph. As you can expect, there are dangers with changing history. (Insert joke about Flash or Harry Potter) We will now see what those dangers are in the next video.

Part 7 - History change using Git

Let's say you branched out a sure feature. After a week, it wasn't so sure anymore. But your commits are good, and you'd like to merge with master. What do you do?

One way would be to discard the commits using a git rebase interactive and then rebase on master, but this is tedious. A simpler command is git rebase onto.

The useless feature can now be deleted safely. What happens to the commits which cannot be reached? They are garbage collected after a time interval. If you want them back, use the git reflog.

Conclusion

Git is a database for content which requires continuous versioning and peer development. It provides wrappers and porcelain commands over this basic feature so that developers can use it to write code as a team.

The principles used here help us understand how systems are built for specific use cases. We can see how important hashing, compression and commit graphs are to its architecture. Now, you can design your own version control system if you'd like to.
Cheers!