# Introduction

It is an introduction to Git that is currently the most widely used version control system in the world, mostly thanks to GitHub. The goal of this guide is to shed some light on how Git works under the hood. It also presents some of the commands (basic and advanced) that you will need to know in order to use Git effectively.

# Understanding Git

## 2 What is Git?

### What is Version Control ?

A version control system is a software designed to keep track of the changes made to files over time. There are a number of benefits to using VCS including the following:

* The ability to undo changes. You can recover an earlier version of you work
* A complete history of all the changes
* Documentation of why changes are made.
* Multiple streams of history.

Working on a team, VCS provides a number of additional benefits

* The ability to resolve conflicts
* Independent streams of history.

One of the most popular VCS tools in use today is called Git. Git is a Distributed VCS, a category known as DVCS, more on that later. Git is free and open source.



Centralized version control



Distributed version control

- There was no longer a central repository, everyone could develop at their own pace, store the updates locally, and put off merging conflicts until their convenience.

- The local nature of DVCSs also made development much faster, since you no longer had to perform actions over a network. And, since each user had a complete copy of the project, the risk of a server crash, a corrupted repository, or any other type of data loss was much lower than that of their CVCS predecessors.

### History of Git

Git is a tool for tracking changes made to a set of files over time, a task traditionally known as “version control.” Although it is most often used by programmers to coordinate changes to software source code, and it is especially good at that, you can use Git to track any kind of content at all. Any body of related files evolving over time, which we’ll call a “project,” is a candidate for using Git. With Git, you can:

* Examine the state of your project at earlier points in time
* Show the differences among various states of the project
* Split the project development into multiple independent lines, called “branches,” which can evolve separately
* Periodically recombine branches in a process called “merging,”

reconciling the changes made in two or more branches

* Allow many people to work on a project simultaneously,

sharing and combining their work as needed

Git is the technology behind the enormously popular “social coding” website GitHub, which includes many wellknown open source projects.

Git is an open source distributed version control system created in 2005 to manage the entire Linux kernel. Instead storing file information in a central repository, Git gives every developer a full copy of the repository

Though originally used for just the Linux kernel, the Git project spread rapidly, and quickly became used to manage a number of other Linux projects,

By far, Git is the most used modern version control system in the world. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel

Having a distributed architecture, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

Git, however, is a distributed version control system. Instead of a working copy, each developer gets their own local repository, complete with a full history of commits

Unlike some version control software, Git is not fooled by the names of the files when determining what the storage and version history of the file tree should be, instead, Git focuses on the file content itself.

Git has been designed with the integrity of managed source code as a top priority. The content of the files as well as the true relationships between files and directories, versions, tags and commits, all of these objects in the Git repository are secured with a cryptographically secure hashing algorithm called SHA1. This protects the code and the change history against both accidental and malicious change and ensures that the history is fully traceable.

With Git, you can be sure you have an authentic content history of your source code.

Some other version control systems have no protections against secret alteration at a later date. This can be a serious information security vulnerability for any organization that relies on software development.

When most SCMs store a new version of a project, they store the code delta or diff. When Git stores a new version of a project, it stores a new *tree* – a bunch of blobs of content and a collection of point­ers that can be expanded back out into a full directory of files and subdirectories. If you want a diff between two versions, it doesn’t add up all the deltas, it simply looks at the two trees and runs a new diff on them.

This is what fundamentally allows the system to be easily distributed – it doesn’t have issues figuring out how to apply a complex series of deltas, it simply transfers all the directories and content that one user has and another does not have but is requesting. It is efficient about it – it only stores identical files and directories once and it can com­press and transfer its content using delta-compressed packfiles – but in concept, is a very simple beast. Git is at it’s heart very stupid-simple.

The tools can be more or less divided into two major camps, often referred to as the *porcelain* and the *plumbing*. The plumbing is not really meant to be used by people on the command line, but rather to do simple things flexibly and are combined by programs and scripts into porcelain programs.

For Windows users, Git installation will install a special command shell called *Git Bash*. To test your installation, open a new command prompt and run

$ git --version.

A version control system is a piece of software that helps the developers on a software team work together and also archives a complete history of their work.

There are three basic goals of a version control system (VCS):

1. We want people to be able to work simultaneously, not serially.
2. When people are working at the same time, we want their changes to not conflict with each other.
3. We want to archive every version of everything that has ever existed — ever.

A filesystem is two-dimensional: Its space is defined by directories and files. In contrast, a repository is threedimensional: It exists in a continuum defined by directories, files, and time. A version control repository contains every version of your source code that has ever existed.

A consequence of this idea is that nothing is ever really destroyed. Every time you make some kind of change to your repository, even if that change is to delete something, the repository gets larger because the history is longer.

Each change adds to the history of the repository. We never subtract anything from that history.

The essential difference between a Centralized Version Control System (CVCS) and a DVCS is the notion of a *repository instance*.

In a CVCS, the repository exists in one place on a central server. Every piece of software that is used to access the repository includes a network client.

Most operations interact with a local repository instance, not a network server. The only time networking code gets involved is when the repository instances are being synchronized. Every developer has his own private repository instance.

In practice, virtually all DVCS teams have a central server. With a CVCS, a central server happens because it is inherent in the centralized model. With a DVCS, a central server happens because of the team’s decision to have one.

## 3 The Git Object Model

We’ll start by inspecting Git’s object database,

Git is a version control system built on top of an *object store*. Git creates and stores a collection of objects when you commit. The object store is stored inside the Git *repository*.

The file-contents reference is actually a reference to a *tree object*. A tree object stores a reference to all the *blob objects* at a particular point in time and other tree objects if there are any subdirectories. A blob object stores the contents of a particular version of a particular single file in the Git repository

## 

## Explore the Object Database

Git keeps all of these objects in the folder .git/objects. This is Git’s object database. Each object, regardless of type, is stored as a file, using its SHA-1 checksum as the filename (sort of). But, instead of storing all objects in a single folder, they are split up using the first two characters of their ID as a directory name, resulting in an object database that looks something like the following.

For example, an object with the following ID:

7a52bb857229f89bffa74134ee3de48e5e146105

is stored in a folder called 7a, using the remaining characters (52bb8...) as a filename. This gives us an object ID, but before we can inspect items in the object database, we need to know what type of object it is. Again, we can use the -t flag:

git cat-file -t 7a52bb8

Of course, change the object ID to an object from your database (don’t forget to combine the folder name with the filename to get the full ID). This will output the type of commit, which we can then pass to a normal call to git cat-file.

git cat-file blob 7a52bb8

My object was a blob, but yours may be different. If it’s a tree, remember to use git ls-tree to turn that ugly binary data into a pretty directory listing.

**git** is fundamentally a key-value store. When you add data to **git**, it builds an object and uses the SHA-1 hash of the object's contents as a key.

A **git** repository is an on-disk data structure which stores metadata for a set of files and directories. It lives in your project's .git**/** folder. Every time you commit data to git, it gets stored here.

It's basic structure is like this:

.git**/**

objects**/**

refs**/**

Therefore, any content in **git** can be looked up by it's hash:

**git cat-file** -p 4bb6f98

There are 4 types of Object:

* Blob
* **Tree**
* Commit
* Tag

HEAD is a special ref. It always points to the current object. You can see where it's currently pointing by checking the .git**/**HEAD file. Normally, HEAD points to another ref:

$cat .git**/**HEAD

ref: refs**/**heads**/**mainline

This is what's known as a "detached head" - because HEAD is not attached to (pointing at) any ref, but rather points directly to an object.

A ref is essentially a pointer. It's a name that points to an object. For example,

"master" --**>** 1a410e...

They are stored in `.git/refs/heads/ in plain text files.

$ **cat** .git**/**refs**/**heads**/**mainline

4bb6f98a223abc9345a0cef9200562333

Now, it's possible to navigate **git** purely by jumping around to different objects directly by their hashes. But this would be terribly inconvenient. A ref gives you a convenient name to refer to objects by. It's much easier to ask **git** to go to a specific place by name rather than by hash.

A commit is probably the object type most familiar to **git** users, as it's what they are used to creating with the **git commit** commands.

However, the commit does not directly contain any changed files or data. Rather, it contains mostly metadata and pointers to other objects which contain the actual contents of the commit.

A commit contains a few things:

* hash of a **tree**
* hash of a parent commit
* author name/email, commiter name/email
* commit message

You can see the contents of any commit like this:

$ **git cat-file** commit 5bac93

**Tree**

A very important note is that the **tree** objects stores EVERY file in your project, and it stores whole files not diffs. This means that each commit contains a snapshot of the entire project\*.

\**Technically, only changed files are stored. But this is more an implementation detail for efficiency. From a design perspective, a commit should be considered as containing a complete copy of the project*.

**Parent**

The parent line contains a hash of another commit object, and can be thought of as a "parent pointer" that points to the "previous commit". This implicitly forms a graph of commits known as the **commit graph**. Specifically, it's a directed acyclic graph (or DAG).

A **tree** basically represents a folder in a traditional filesystem: nested containers for files or other folders.

A **tree** contains:

* 0 or more blob objects
* 0 or more **tree** objects

A blob contains arbitrary binary file contents. Commonly, it will be raw text such as source code or a blog article. But it could just as easily be the bytes of a PNG file or anything else.

If you have the hash of a blob, you can look at it's contents.

The **git commit** command does a few things:

1. Create blobs and trees to represent your project directory - stored in .git**/**objects

2. Creates a new commit object with your author information, commit message, and the root **tree** from step 1 - also stored in .git**/**objects

3. Updates the HEAD ref in .git**/**HEAD to the hash of the newly-created commit

This results in a new snapshot of your project being added to **git** that is connected to the previous state.

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects. The objects are **commits**, **trees**, **blobs,** and **tags**. These objects are stored in **.git/objects**.

>ls -al

A commit object stores the hash of the directory tree object that the commit corresponds to, the parent commit hash, the author, the committer date and time, and the commit message:

**git cat-file -t 11b8b15**

**git cat-file -p 11b8b15**

**BLOBs**

Git uses blobs to store the contents of a file at a given point in time. A blob is a **Binary Large OBject (BLOB)**. It's Git's methodology of storing the contents of a file at a given point in its lifetime. A blob is created when we commence the tracking of a file by using the **git add** command:

Git uses objects to track changes throughout the history of a repository. To achieve this tracking, Git uses four types of objects. The objects are **commits**, **trees**, **blobs,** and **tags**. These objects are stored in **.git/objects**.

### The SHA

All the information needed to represent the history of a project is stored in files referenced by a 40-digit "object name" that looks something like this:

6ff87c4664981e4397625791c8ea3bbb5f2279a3

The SHA1 hash is a cryptographic hash function. What that means to us is that it is virtually impossible to find two different objects with the same name. This has a number of advantages; among others:

* Git can quickly determine whether two objects are identical or not, just by comparing names.
* Since object names are computed the same way in every repository, the same content stored in two repositories will always be stored under the same name.
* Git can detect errors when it reads an object, by checking that the object's name is still the SHA1 hash of its contents

### The Objects

Every object consists of three things - a **type**, a **size** and **content**. The *size* is simply the size of the contents, the contents depend on what type of object it is, and there are four different types of objects: "blob", "tree", "commit", and "tag".

* A **"blob"** is used to store file data - it is generally a file.
* A **"tree"** is basically like a directory - it references a bunch of other trees and/or blobs (i.e. files and sub-directories)
* A **"commit"** points to a single tree, marking it as what the project looked like at a certain point in time. It contains meta-information about that point in time, such as a timestamp, the author of the changes since the last commit, a pointer to the previous commit(s), etc.
* A **"tag"** is a way to mark a specific commit as special in some way. It is normally used to tag certain commits as specific releases or something along those lines.

Almost all of Git is built around manipulating this simple structure of four different object types. It is sort of its own little filesystem that sits on top of your machine's filesystem.

It is important to note that this is very different from most SCM systems that you may be familiar with. Subversion, CVS, Perforce, Mercurial and the like all use Delta Storage systems - they store the differences between one commit and the next. Git does not do this - it stores a snapshot of what all the files in your project look like in this tree structure each time you commit. This is a very important concept to understand when using Git.

### Blob Object

A blob generally stores the contents of a file.

You can use git-show to examine the contents of any blob. Since the blob is entirely defined by its data, if two files in a directory tree (or in multiple different versions of the repository) have the same contents, they will share the same blob object. The object is totally independent of its location in the directory tree, and renaming a file does not change the object that file is associated with

### Tree Object

A tree is a simple object that has a bunch of pointers to blobs and other trees - it generally represents the contents of a directory or subdirectory.

The ever-versatile git-show command can also be used to examine tree objects, but :git-ls-tree will give you more details.

### Commit Object

The "commit" object links a physical state of a tree with a description of how we got there and why.

You can use the --pretty=raw option to git-show or git-log to examine your favorite commit.

A commit is defined by:

* a **tree**: The SHA1 name of a tree object (as defined below), representing the contents of a directory at a certain point in time.
* **parent(s)**: The SHA1 name of some number of commits which represent the immediately previous step(s) in the history of the project. The example above has one parent; merge commits may have more than one. A commit with no parents is called a "root" commit, and represents the initial revision of a project. Each project must have at least one root. A project can also have multiple roots, though that isn't common (or necessarily a good idea).
* an **author**: The name of the person responsible for this change, together with its date.
* a **committer**: The name of the person who actually created the commit, with the date it was done. This may be different from the author; for example, if the author wrote a patch and emailed it to another person who used the patch to create the commit.
* a **comment** describing this commit.

A commit is usually created by git-commit, which creates a commit whose parent is normally the current HEAD, and whose tree is taken from the content currently stored in the index.

### Tag Object

A tag object contains an object name (called simply 'object'), object type, tag name, the name of the person ("tagger") who created the tag, and a message, as can be seen using git-cat-file.

$ git cat-file tag v1.5.0

git-tag can also be used to create "lightweight tags", which are not tag objects at all, but just simple references whose names begin with "refs/tags/"

## Git Directory and Working Directory

### The Git Directory

The 'git directory' is the directory that stores all Git's history and meta information for your project - including all of the objects (commits, trees, blobs, tags), all of the pointers to where different branches are and more.

There is only one Git Directory per project (as opposed to one per subdirectory like with SVN or CVS), and that directory is (by default, though not necessarily) '.git' in the root of your project. If you look at the contents of that directory, you can see all of your important files:

$>tree -L 1

.

|-- HEAD # pointer to your current branch

|-- config # your configuration preferences

|-- description # description of your project

|-- hooks/ # pre/post action hooks

|-- index # index file (see next section)

|-- logs/ # a history of where your branches have been

|-- objects/ # your objects (commits, trees, blobs, tags)

`-- refs/ # pointers to your branches

### The Working Directory

The Git 'working directory' is the directory that holds the current checkout of the files you are working on. Files in this directory are often removed or replaced by Git as you switch branches - this is normal. All your history is stored in the Git Directory; the working directory is simply a temporary checkout place where you can modify the files until your next commit.

## The Git Index

The Git index is used as a staging area between your working directory and your repository. You can use the index to build up a set of changes that you want to commit together. When you create a commit, what is committed is what is currently in the index, not what is in your working directory.

The easiest way to see what is in the index is with the git-status command. When you run git status, you can see which files are staged (currently in your index), which are modified but not yet staged, and which are completely untracked.

## How Git Stores Objects

All objects are stored as compressed contents by their sha values.

If the sha of your object is ab04d884140f7b0cf8bbf86d6883869f16a46f65, then the file will be stored in the following path:

.git/objects/ab/04d884140f7b0cf8bbf86d6883869f16a46f65

It pulls the first two characters off and uses that as the subdirectory, so that there are never too many objects in one directory. The actual file name is the remaining 38 characters.

## Browsing Git Objects

We can ask git about particular objects with the cat-file command. Note that you can shorten the shas to only a few characters to save yourself typing all 40 hex digits:

$ git cat-file -t 54196cc2

$ git cat-file -s 54196cc2

$ git cat-file -p 54196cc2

You can examine the contents of any tree using ls-tree

$ git ls-tree 92b8b694

All of these objects are stored under their SHA1 names inside the git directory and the contents of these files is just the compressed data plus a header identifying their length and their type. The type is either a blob, a tree, a commit, or a tag.

$ find .git/objects/

The simplest commit to find is the HEAD commit, which we can find from .git/HEAD:

$ cat .git/HEAD

ref: refs/heads/master

$ cat .git/refs/heads/master

c4d59f390b9cfd4318117afde11d601c1085f241

$ git cat-file -t c4d59f39

commit

The "tree" object here refers to the new state of the tree and the "parent" object refers to the previous commit:

## Git References

Branches, remote-tracking branches, and tags are all references to commits. All references are named with a slash-separated path name starting with "refs"; the names we've been using so far are actually shorthand:

- The branch "test" is short for "refs/heads/test".

- The tag "v2.6.18" is short for "refs/tags/v2.6.18".

- "origin/master" is short for "refs/remotes/origin/master

We can list all the heads in this repository with linkgit:git-show-ref

$ git show-ref --heads

$ git show-ref --tags

## The Git Index

The index is a binary file (generally kept in .git/index) containing a sorted list of path names, each with permissions and the SHA1 of a blob object; linkgit:git-ls-files[1] can show you the contents of the index:

$ git ls-files --stage

Computes the object ID value for an object with specified type with the contents of the named file

$ git hash-objec

## [Git from the inside out](https://maryrosecook.com/blog/post/git-from-the-inside-out)

The .git directory and its contents are Git’s. All the other files are collectively known as the working copy. They are the user’s.

### Add some files

$ git add data/letter.txt

This command has two effects.

* First, it creates a new blob file in the .git/objects/ directory.

For example, Git hashes the content to 2e65efe2a145dda7ee51d1741299f848e5bf752e. The first two characters are used as the name of a directory inside the objects database: .git/objects/2e/. The rest of the hash is used as the name of the blob file that holds the content of the added file: .git/objects/2e/65efe2a145dda7ee51d1741299f848e5bf752e.

* Second, git add adds the file to the index. The index is a list that contains every file that Git has been told to keep track of. It is stored as a file at .git/index.

### Make a commit

git commit -m 'a1'

The commit command has three steps. It creates a tree graph to represent the content of the version of the project being committed. It creates a commit object. It points the current branch at the new commit object

* Git records the current state of the project by creating a tree graph from the index. This tree graph records the location and content of every file in the project.

The graph is composed of two types of object: blobs and trees.

Blobs are stored by git add. They represent the content of files.

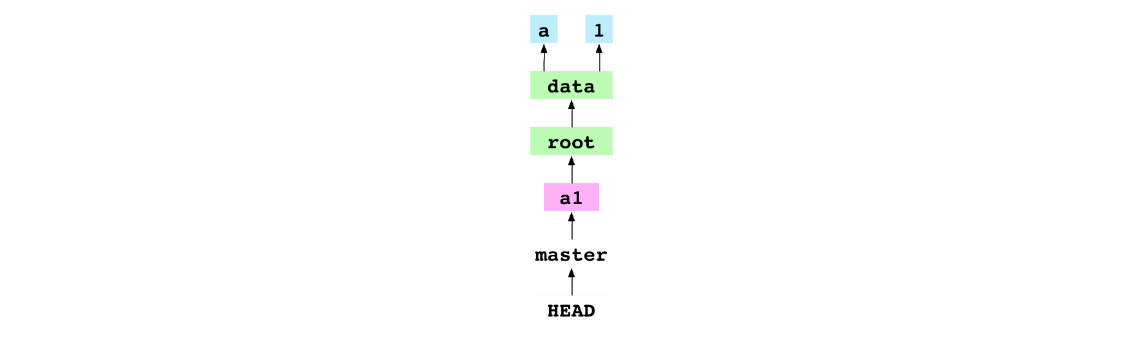
Trees are stored when a commit is made. A tree represents a directory in the working copy.

* git commit creates a commit object after creating the tree graph. The commit object is just another text file in .git/objects/:
* Finally, the commit command points the current branch at the new commit object. Which is the current branch? Git goes to the HEAD file at .git/HEAD and finds:

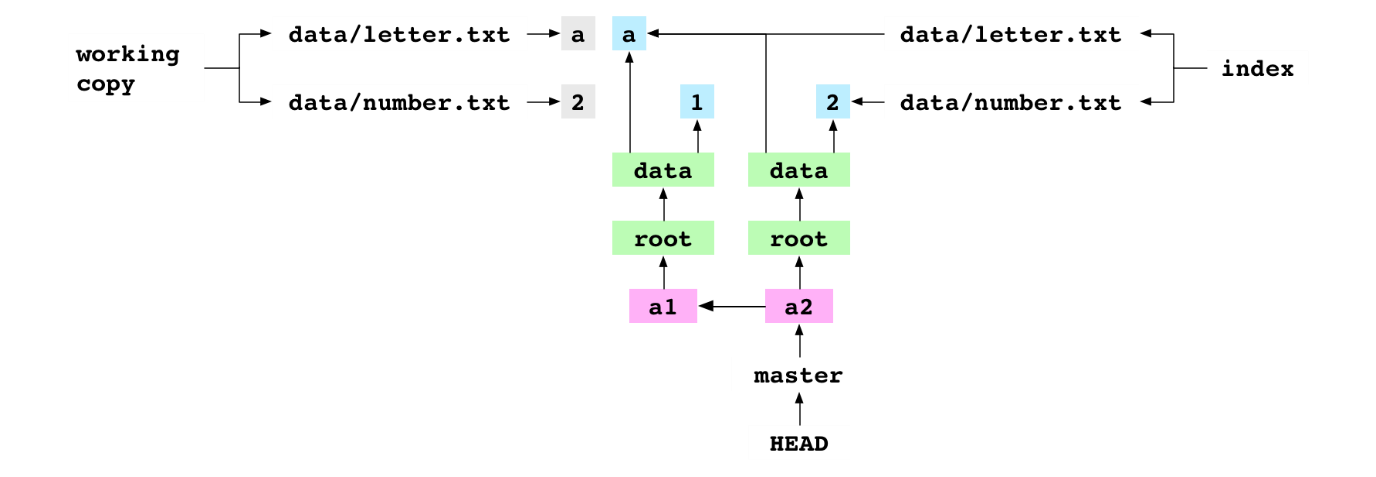
ref: refs/heads/master

This says that HEAD is pointing at master. master is the current branch.

HEAD and master are both refs. A ref is a label used by Git or the user to identify a specific commit.



HEAD pointing at master and master pointing at the a1 commit



### Check out a commit

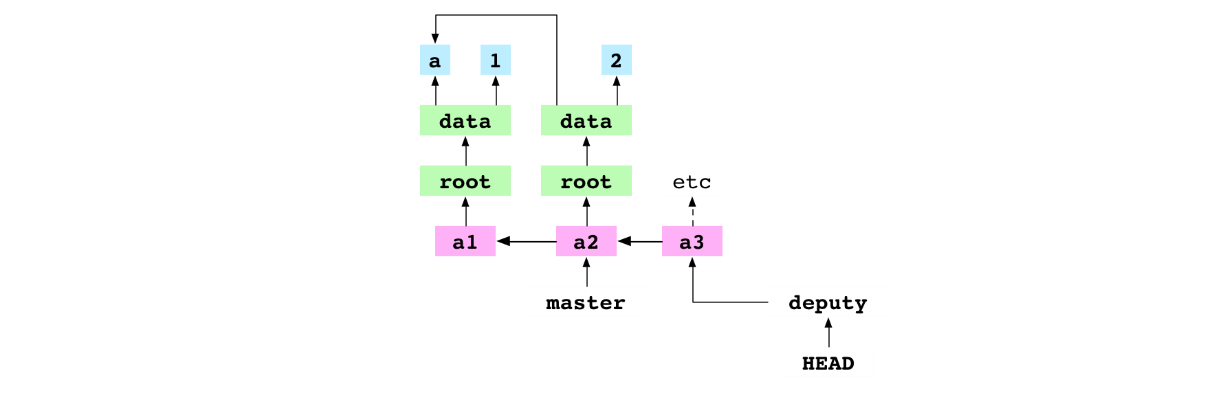
git checkout 37888c2

### Create a branch

git branch deputy

The user creates a new branch called deputy. This just creates a new file at .git/refs/heads/deputy that contains the hash that HEAD is pointing at: the hash of the a3 commit.

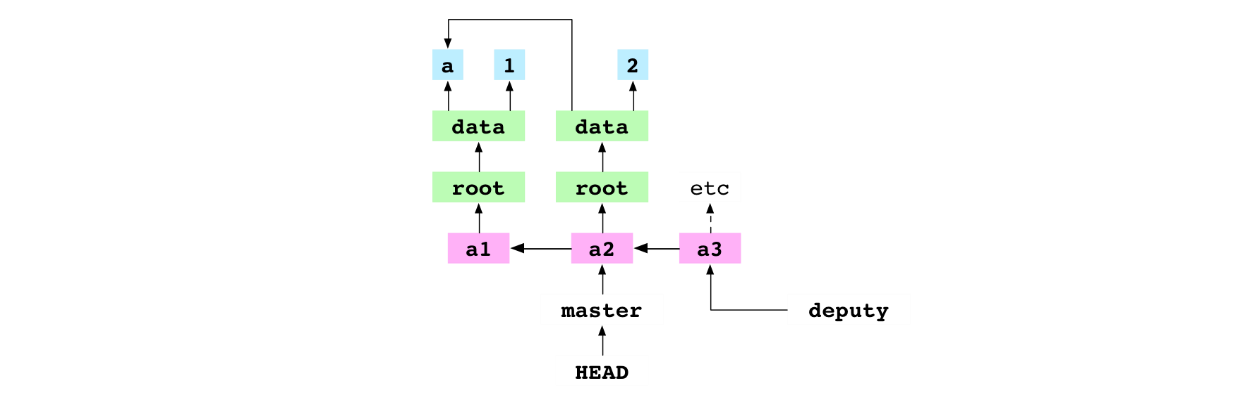
### Merge an ancestor



git merge master

Merging two branches means merging two commits. The first commit is the one that deputy points at: the receiver. The second commit is the one that master points at: the giver. For this merge, Git does nothing. It reports it is Already up-to-date.

### Merge a descendent



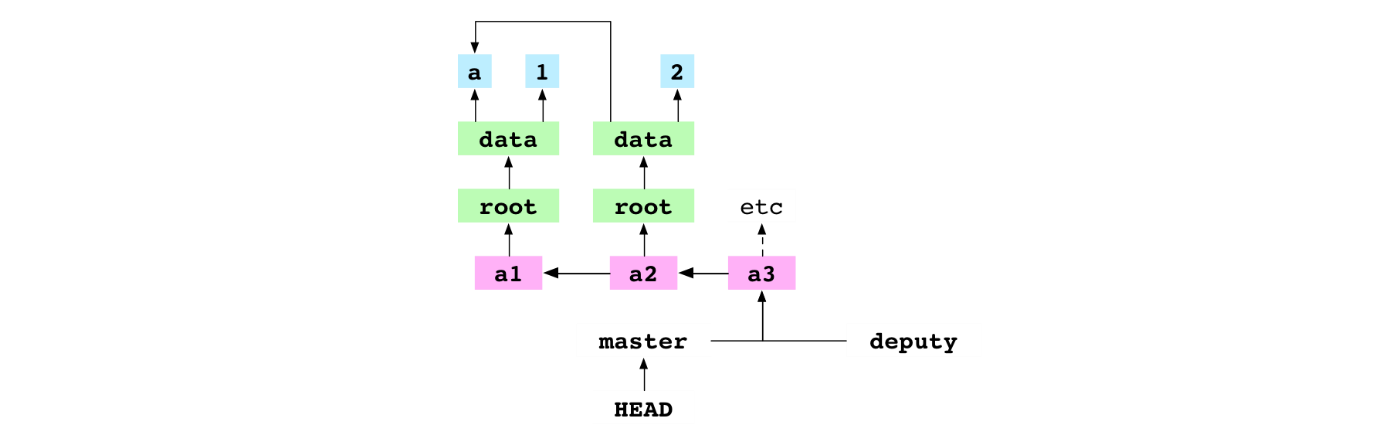
master checked out and pointing at the a2 commit

git checkout master

git merge deputy

They merge deputy into master. Git discovers that the receiver commit, a2, is an ancestor of the giver commit, a3. It can do a fast-forward merge.

It gets the giver commit and gets the tree graph that it points at. It writes the file entries in the tree graph to the working copy and the index. It “fast-forwards” master to point at a3.



a3 commit from deputy fast-forward merged into master

### Merge two commits from different lineages

$ printf '4' > data/number.txt

$ git add data/number.txt

$ git commit -m 'a4'

The user sets the content of number.txt to 4 and commits the change to master.

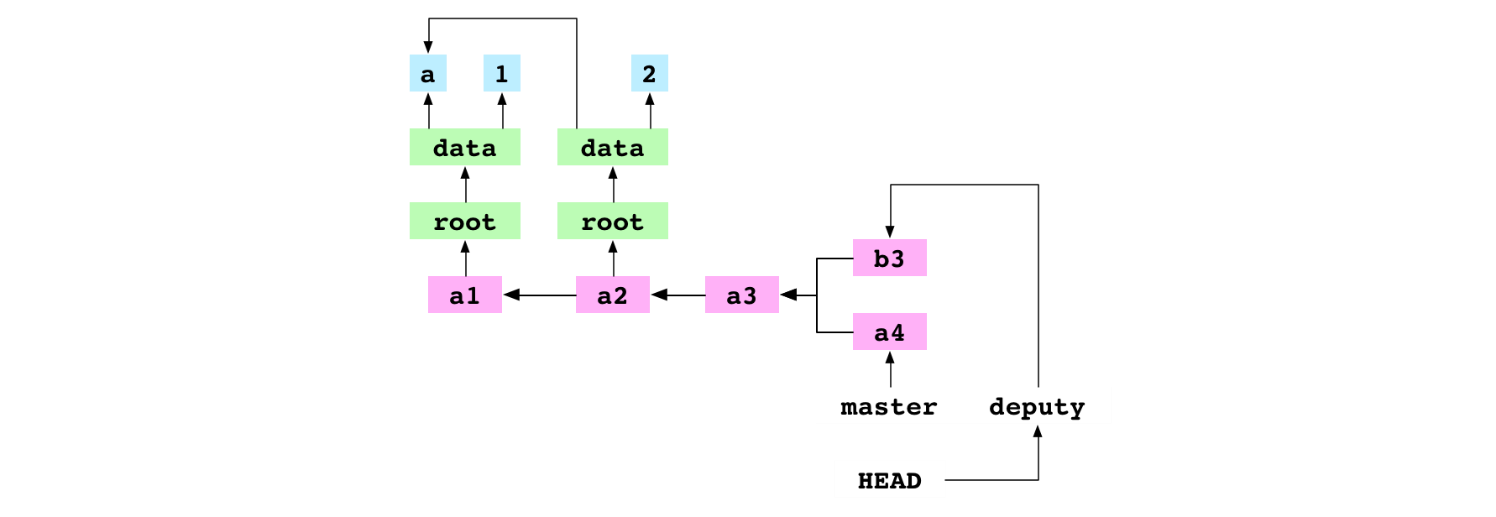
$ git checkout deputy

$ printf 'b' > data/letter.txt

$ git add data/letter.txt

$ git commit -m 'b3'

The user checks out deputy. They set the content of data/letter.txt to b and commit the change to deputy.



a4 committed to master, b3 committed to deputy and deputy checked out

The user merges master into deputy.

git merge master -m 'b4'

Git discovers that the receiver, b3, and the given, a4, are in different lineages. It makes a merge commit. This process has eight steps.

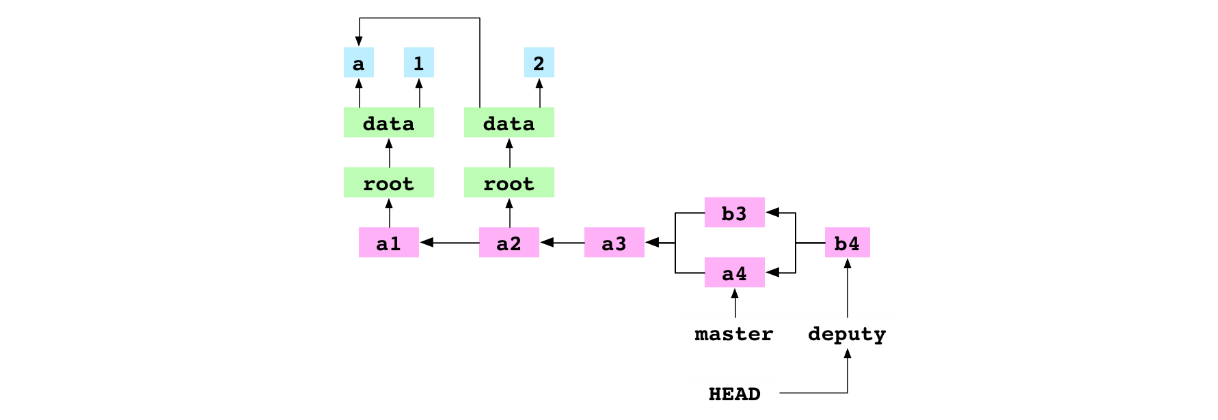
* First, Git writes the hash of the giver commit to a file at alpha/.git/MERGE\_HEAD. The presence of this file tells Git it is in the middle of merging.
* Second, Git finds the base commit: the most recent ancestor that the receiver and giver commits have in common.
* Third, Git generates the indices for the base, receiver and giver commits from their tree graphs.
* Fourth, Git generates a diff that combines the changes made to the base by the receiver commit and the giver commit. This diff is a list of file paths that point to a change: add, remove, modify or conflict.

Git gets the list of all the files that appear in the base, receiver or giver indices. For each one, it compares the index entries to decide the change to make to the file. It writes a corresponding entry to the diff. In this case, the diff has two entries.

The first entry is for data/letter.txt. The content of this file is a in the base, b in the receiver and a in the giver. The content is different in the base and receiver. But it is the same in the base and giver. Git sees that the content was modified by the receiver, but not the giver. The diff entry for data/letter.txt is a modification, not a conflict.

The second entry in the diff is for data/number.txt. In this case, the content is the same in the base and receiver, and different in the giver. The diff entry for data/letter.txt is also a modification.

* Fifth, the changes indicated by the entries in the diff are applied to the working copy. The content of data/letter.txt is set to b and the content of data/number.txt is set to 4.
* Sixth, the changes indicated by the entries in the diff are applied to the index. The entry for data/letter.txt is pointed at the b blob and the entry for data/number.txt is pointed at the 4 blob
* Seventh, the updated index is committed
* Eighth, Git points the current branch, deputy, at the new commit.



Notice that the commit has two parents.

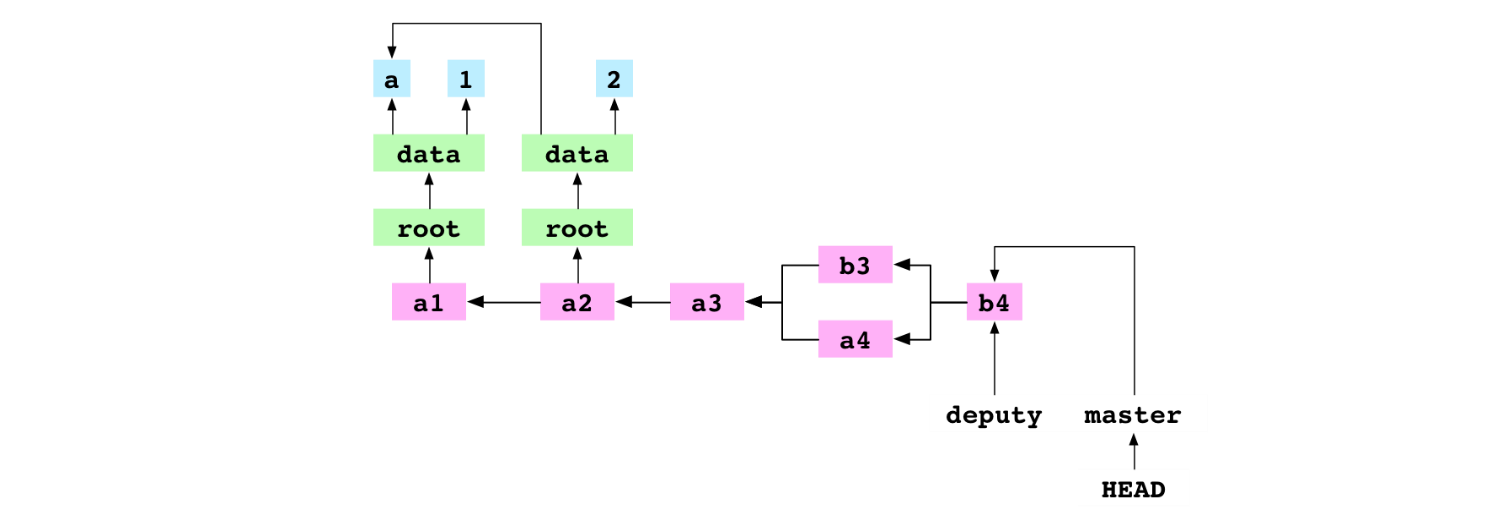
b4, the merge commit resulting from the recursive merge of a4 into b3

### Merge two commits from different lineages that both modify the same file

$ git checkout master

$ git merge deputy

The user checks out master. They merge deputy into master. This fast-forwards master to the b4 commit. master and deputy now point at the same commit.



deputy merged into master to bring master up to the latest commit, b4

The user checks out deputy. They set the content of data/number.txt to 5 and commit the change to deputy.

$ git checkout deputy

$ printf '5' > data/number.txt

$ git add data/number.txt

$ git commit -m 'b5'

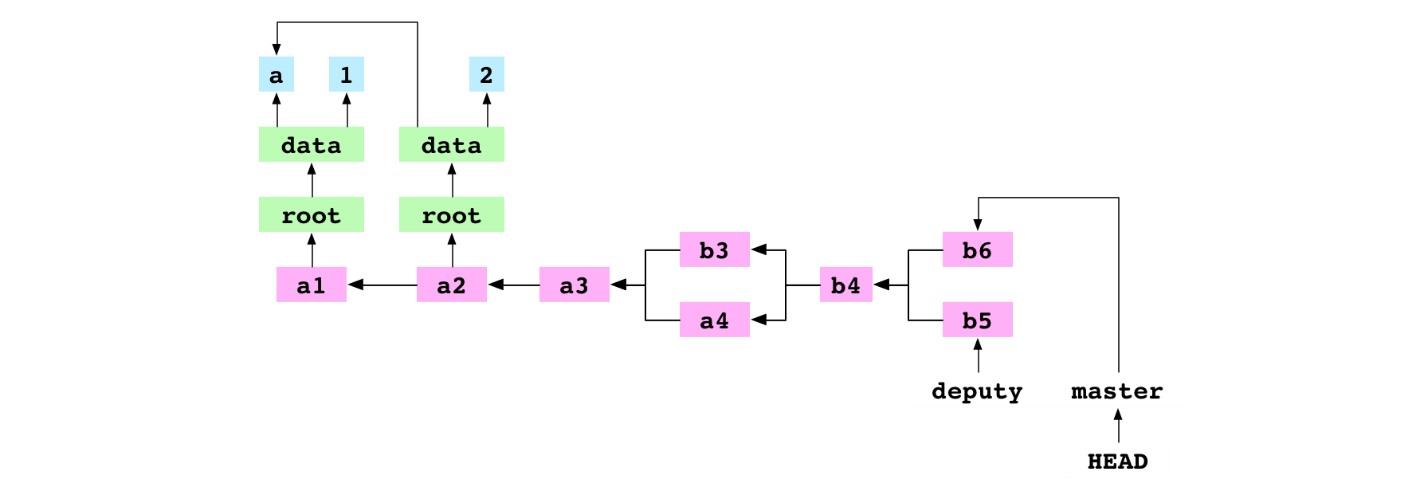
The user checks out master. They set the content of data/number.txt to 6 and commit the change to master.

$ git checkout master

$ printf '6' > data/number.txt

$ git add data/number.txt

$ git commit -m 'b6'



b5 commit on deputy and b6 commit on master

$ git merge deputy

The user merges deputy into master. There is a conflict and the merge is paused.

The user merges deputy into master. There is a conflict and the merge is paused. The process for a conflicted merge follows the same first six steps as the process for an unconflicted merge:

set .git/MERGE\_HEAD,

find the base commit,

generate the indices of the base, receiver and giver commits,

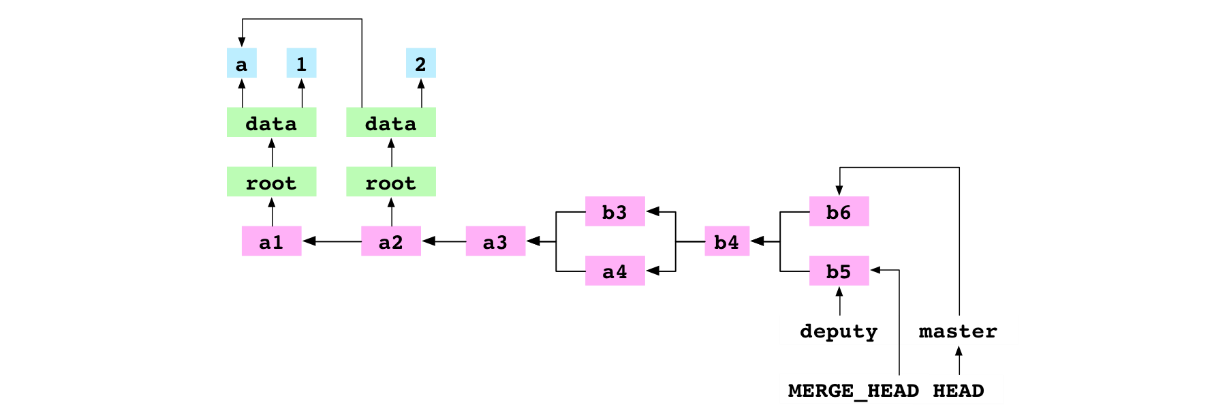
create a diff,

update the working copy

update the index.

Because of the conflict, the seventh commit step and eighth ref update step are never taken. Let’s go through the steps again and see what happens.

First, Git writes the hash of the giver commit to a file at .git/MERGE\_HEAD



MERGE\_HEAD written during merge of b5 into b6

Second, Git finds the base commit, b4.

Third, Git generates the indices for the base, receiver and giver commits.

Fourth, Git generates a diff that combines the changes made to the base by the receiver commit and the giver commit. This diff is a list of file paths that point to a change: add, remove, modify or conflict.

In this case, the diff contains only one entry: data/number.txt. The entry is marked as a conflict because the content for data/number.txt is different in the receiver, giver and base.

Fifth, the changes indicated by the entries in the diff are applied to the working copy. For a conflicted area, Git writes both versions to the file in the working copy. The content of data/number.txt is set to:

<<<<<<< HEAD

6

=======

5

>>>>>>> deputy

Sixth, the changes indicated by the entries in the diff are applied to the index. Entries in the index are uniquely identified by a combination of their file path and stage. The entry for an unconflicted file has a stage of 0. Before this merge, the index looked like this, where the 0s are stage values:

0 data/letter.txt 63d8dbd40c23542e740659a7168a0ce3138ea748

0 data/number.txt 62f9457511f879886bb7728c986fe10b0ece6bcb

After the merge diff is written to the index, the index looks like this:

0 data/letter.txt 63d8dbd40c23542e740659a7168a0ce3138ea748

1 data/number.txt bf0d87ab1b2b0ec1a11a3973d2845b42413d9767

2 data/number.txt 62f9457511f879886bb7728c986fe10b0ece6bcb

3 data/number.txt 7813681f5b41c028345ca62a2be376bae70b7f61

he entry for data/letter.txt at stage 0 is the same as it was before the merge. The entry for data/number.txt at stage 0 is gone. There are three new entries in its place. The entry for stage 1 has the hash of the base data/number.txt content. The entry for stage 2 has the hash of the receiver data/number.txt content. The entry for stage 3 has the hash of the giver data/number.txt content. The presence of these three entries tells Git that data/number.txt is in conflict.

The merge pauses

~/alpha $ printf '11' > data/number.txt

~/alpha $ git add data/number.txt

The user integrates the content of the two conflicting versions by setting the content of data/number.txt to 11. They add the file to the index. Git adds a blob containing 11. Adding a conflicted file tells Git that the conflict is resolved. Git removes the data/number.txt entries for stages 1, 2 and 3 from the index. It adds an entry for data/number.txt at stage 0 with the hash of the new blob. The index now reads:

0 data/letter.txt 63d8dbd40c23542e740659a7168a0ce3138ea748

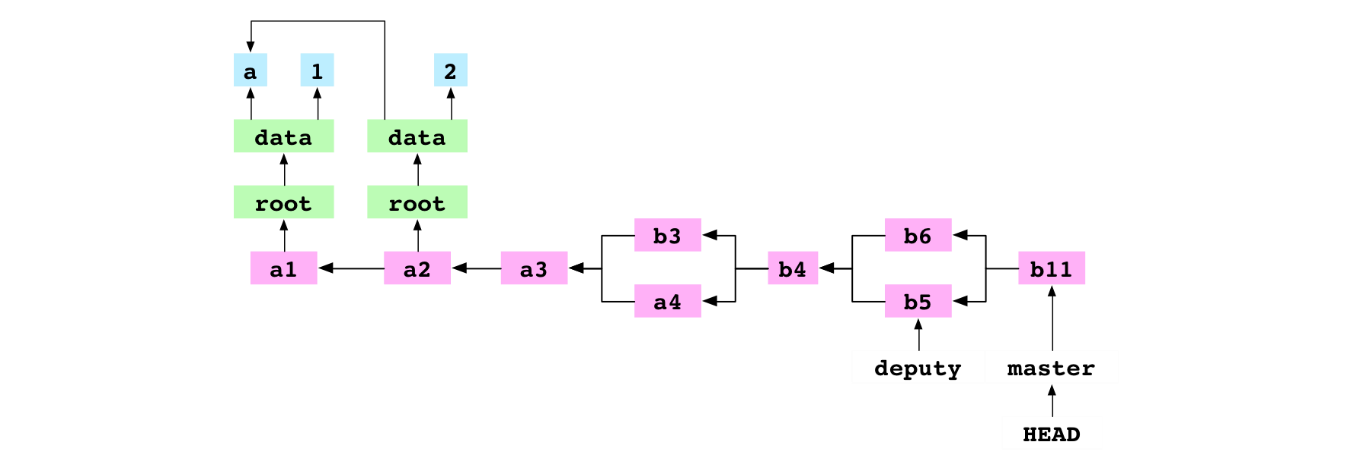
0 data/number.txt 9d607966b721abde8931ddd052181fae905db503

Seventh, the user commits.

git commit -m 'b11'

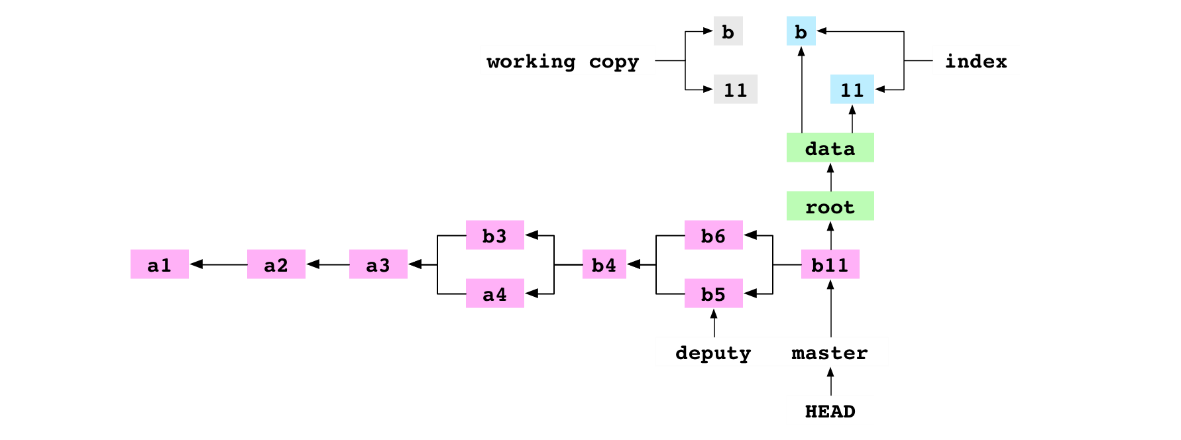
Git sees .git/MERGE\_HEAD in the repository, which tells it that a merge is in progress. It checks the index and finds there are no conflicts. It creates a new commit, b11, to record the content of the resolved merge. It deletes the file at .git/MERGE\_HEAD. This completes the merge.

Eighth, Git points the current branch, master, at the new commit



b11, the merge commit resulting from the conflicted, recursive merge of b5 into b6

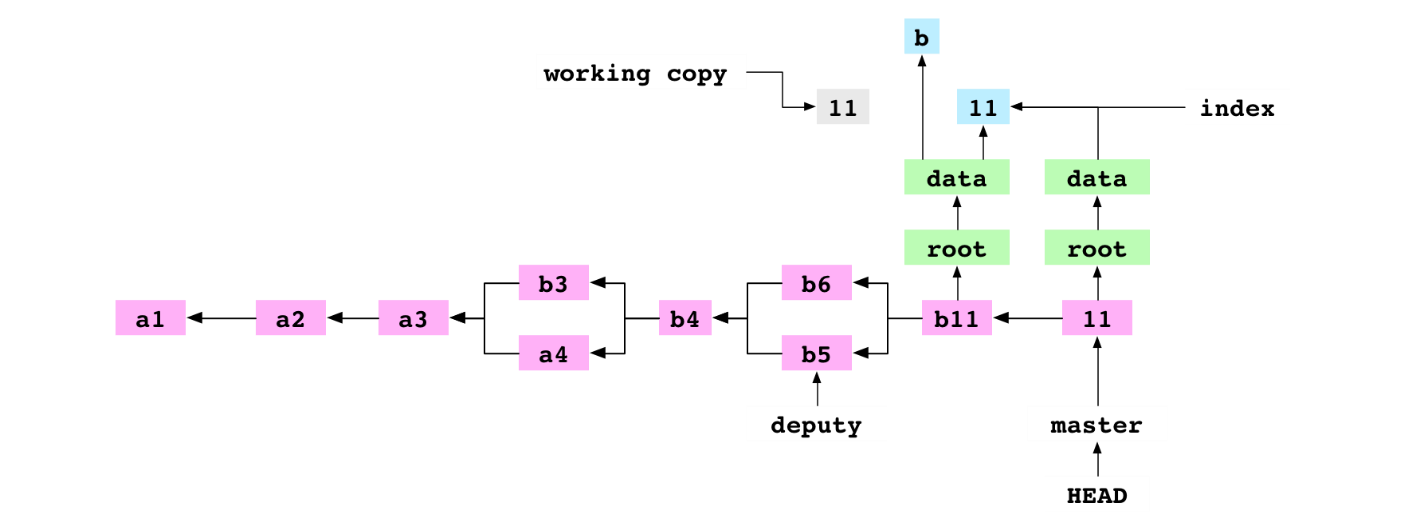
### Remove a file



git rm data/letter.txt

The user tells Git to remove data/letter.txt. The file is deleted from the working copy. The entry is deleted from the index

git commit -m '11'



11 commit made after data/letter.txt rm ed

### Copy a repository

~/alpha $ cd ..

~ $ cp -R alpha bravo

The user copies the contents of the alpha/ repository to the bravo/ directory. This produces the following directory structure:

├── alpha

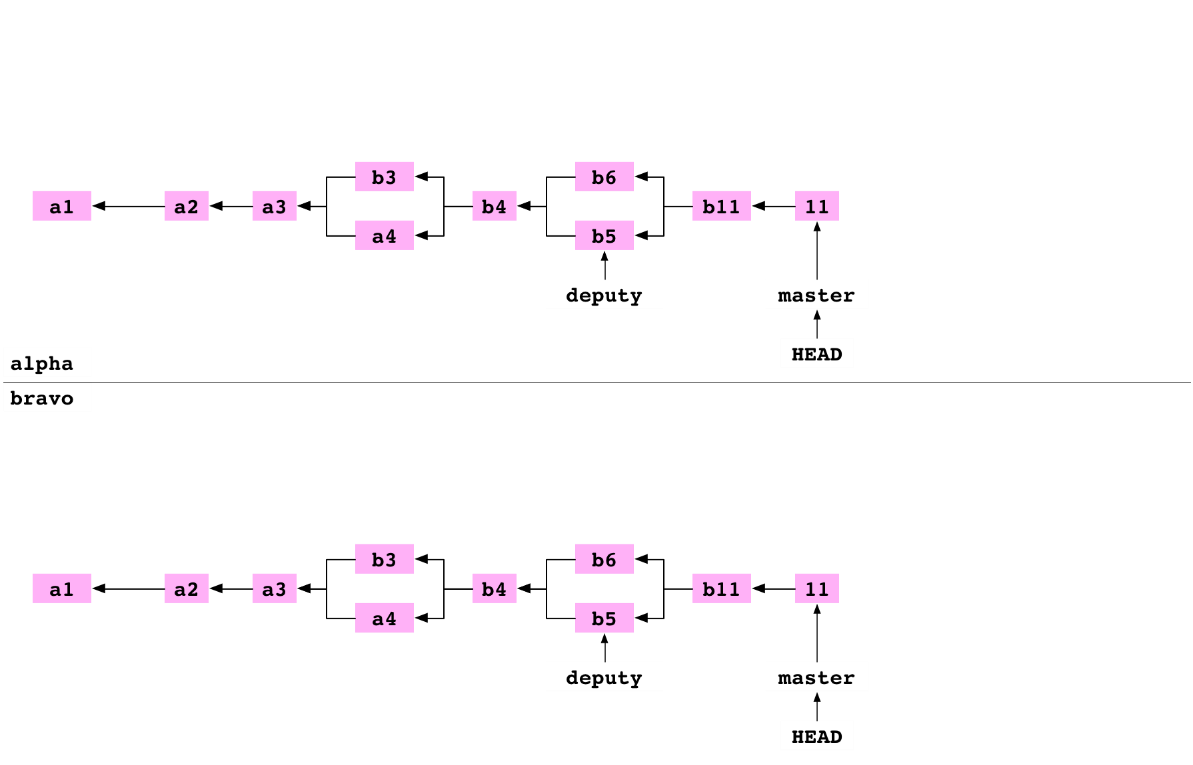
| └── data

| └── number.txt

└── bravo

└── data

└── number.txt



New graph created when alpha cp ed to bravo

### Link a repository to another repository

~ $ cd alpha

~/alpha $ git remote add bravo ../bravo

The user moves back into the alpha repository. They set up bravo as a remote repository on alpha. This adds some lines to the file at alpha/.git/config:

[remote "bravo"]

url = ../bravo/

These lines specify that there is a remote repository called bravo in the directory at ../bravo.

### Fetch a branch from a remote

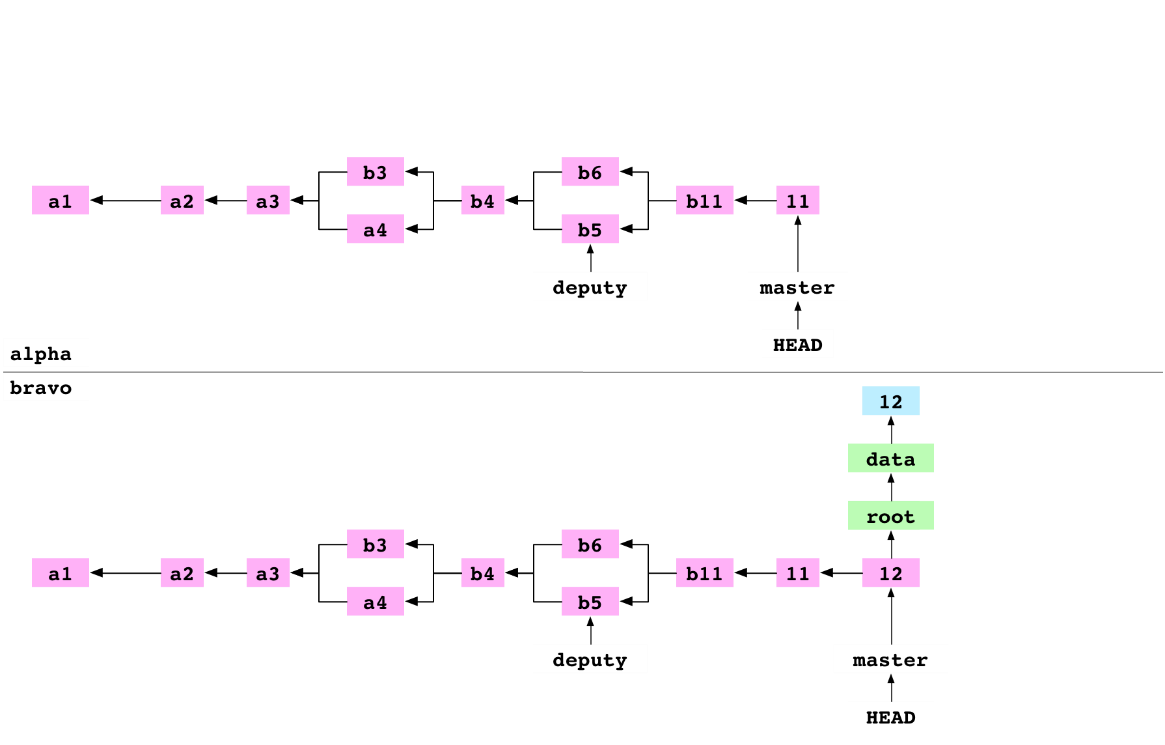
~/alpha $ cd ../bravo

~/bravo $ printf '12' > data/number.txt

~/bravo $ git add data/number.txt

~/bravo $ git commit -m '12'

The user goes into the bravo repository. They set the content of data/number.txt to 12 and commit the change to master on bravo.



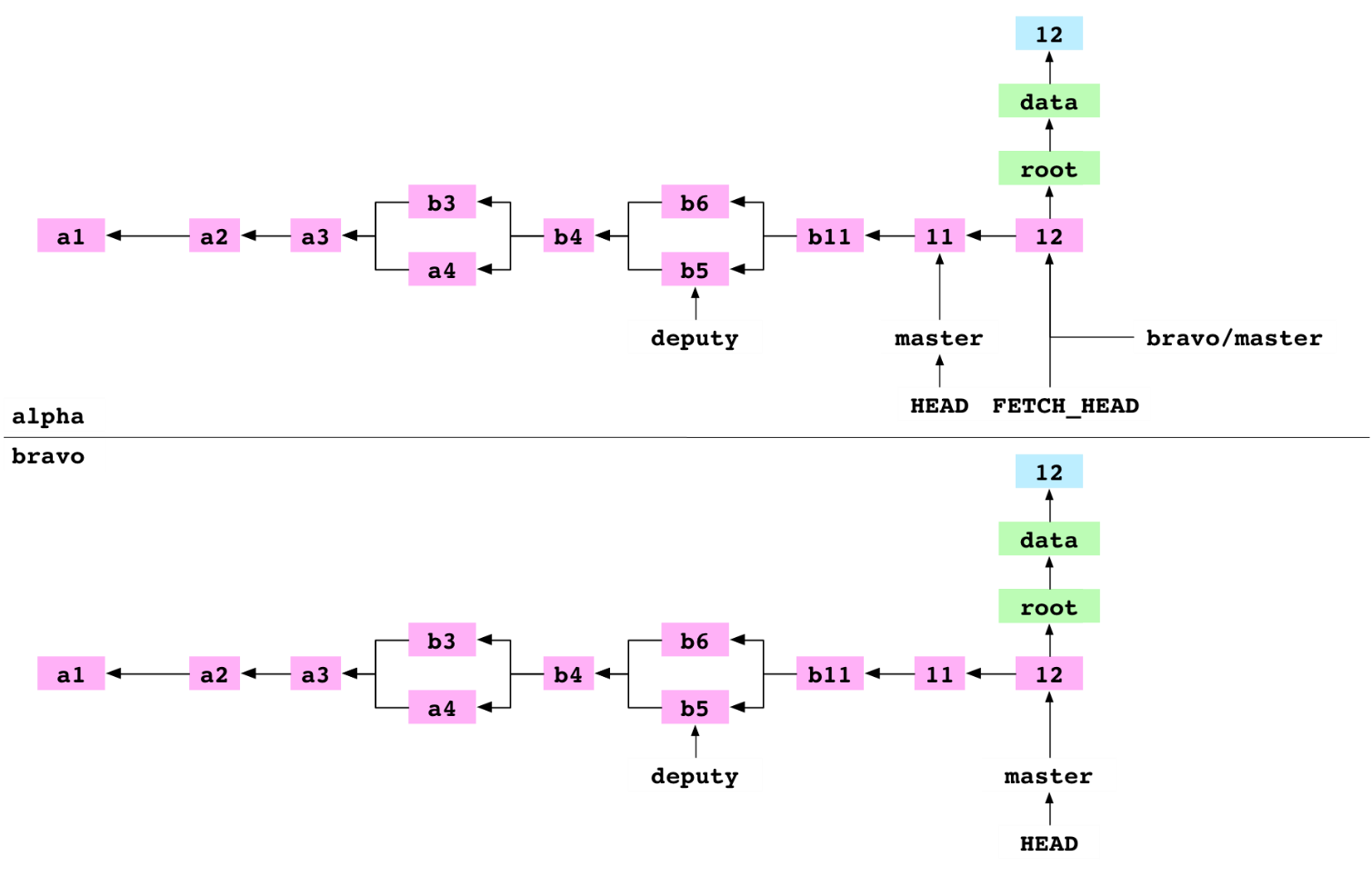
~/bravo $ cd ../alpha

~/alpha $ git fetch bravo master

The user goes into the alpha repository. They fetch master from bravo into alpha. This process has four steps.

* First, Git gets the hash of the commit that master is pointing at on bravo. This is the hash of the 12 commit.
* Second, Git makes a list of all the objects that the 12 commit depends on: the commit object itself, the objects in its tree graph, the ancestor commits of the 12 commit and the objects in their tree graphs. It removes from this list any objects that the alpha object database already has. It copies the rest to alpha/.git/objects/.
* Third, the content of the concrete ref file at alpha/.git/refs/remotes/bravo/master is set to the hash of the 12 commit.
* Fourth, the content of alpha/.git/FETCH\_HEAD is set to:

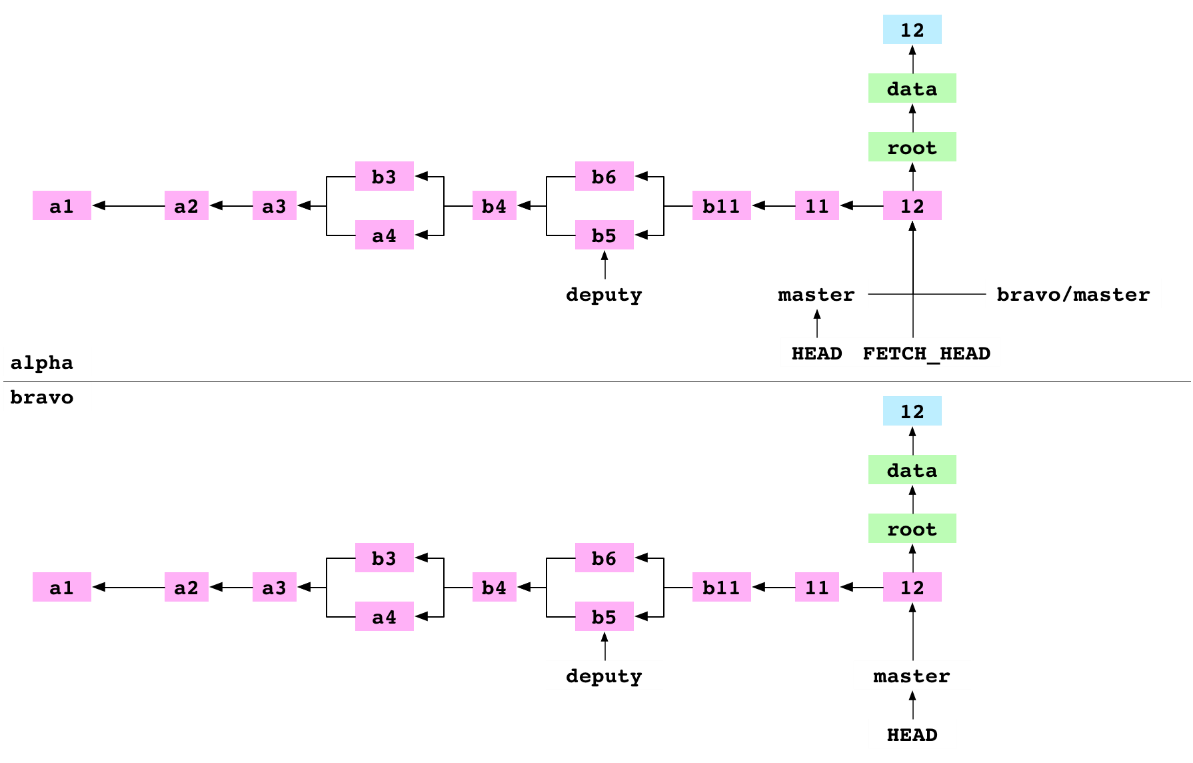
This indicates that the most recent fetch command fetched the 12 commit of master from bravo.



### Merge FETCH\_HEAD

git merge FETCH\_HEAD

The user merges FETCH\_HEAD. FETCH\_HEAD is just another ref. It resolves to the 12 commit, the giver. HEAD points at the 11 commit, the receiver. Git does a fast-forward merge and points master at the 12 commit.



alpha after FETCH\_HEAD merged

### Pull a branch from a remote

git pull bravo master

The user pulls master from bravo into alpha. Pull is shorthand for “fetch and merge FETCH\_HEAD”. Git does these two commands and reports that master is Already up-to-date

### Clone a repository

~/alpha $ cd ..

~ $ git clone alpha charlie

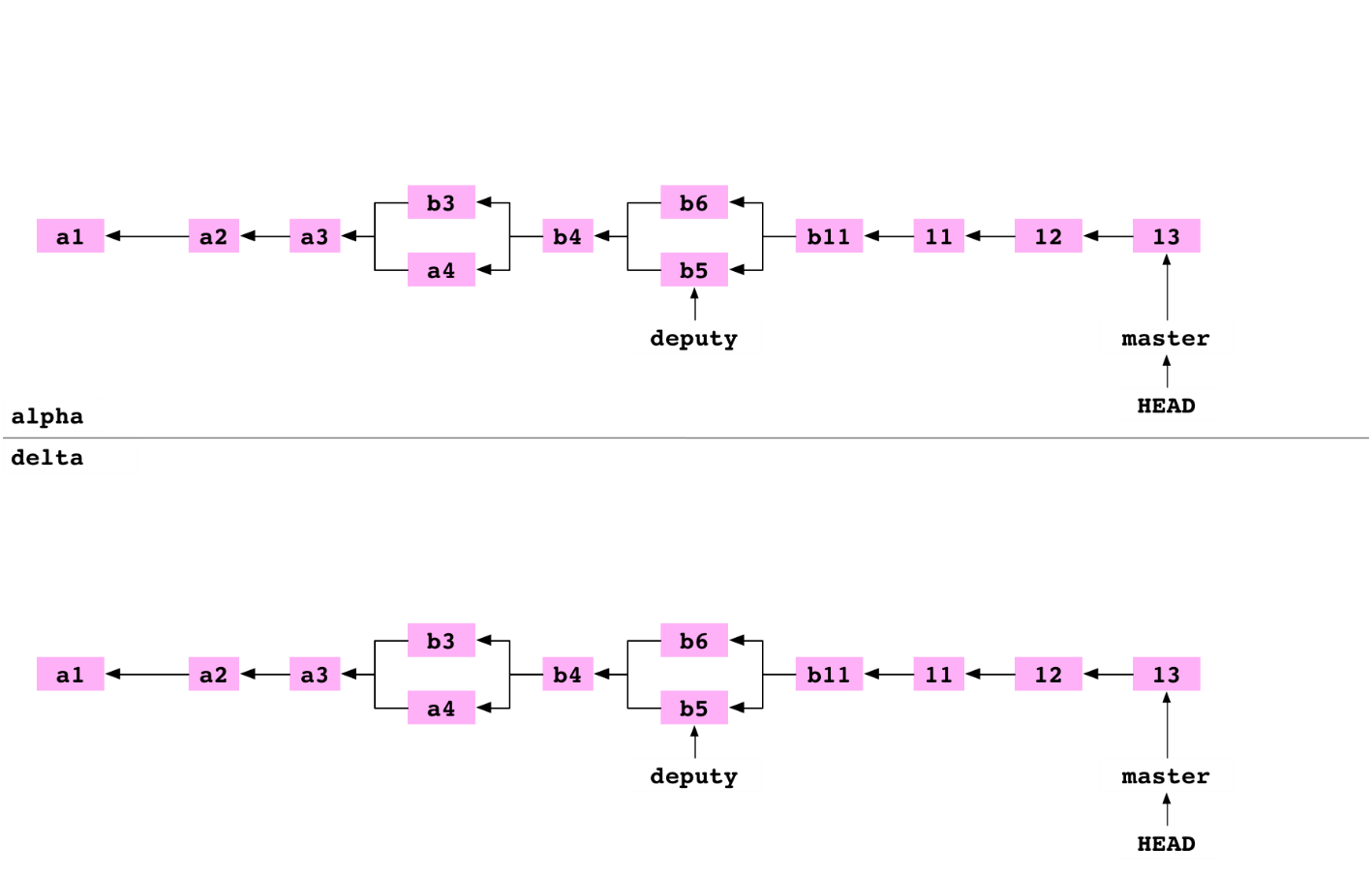
The user moves into the directory above. They clone alpha to charlie. Cloning to charlie has similar results to the cp the user did to produce the bravo repository. Git creates a new directory called charlie. It inits charlie as a Git repo, adds alpha as a remote called origin, fetches origin and merges FETCH\_HEAD.

### Clone a bare repository

/alpha $ cd ..

~ $ git clone alpha delta --bare

Cloning into bare repository 'delta'



The user moves into the directory above. They clone delta as a bare repository. This is an ordinary clone with two differences. The config file indicates that the repository is bare. And the files that are normally stored in the .git directory are stored in the root of the repository:

### Push a branch to a bare repository

The user goes back into the alpha repository. They set up delta as a remote repository on alpha.

~ $ cd alpha

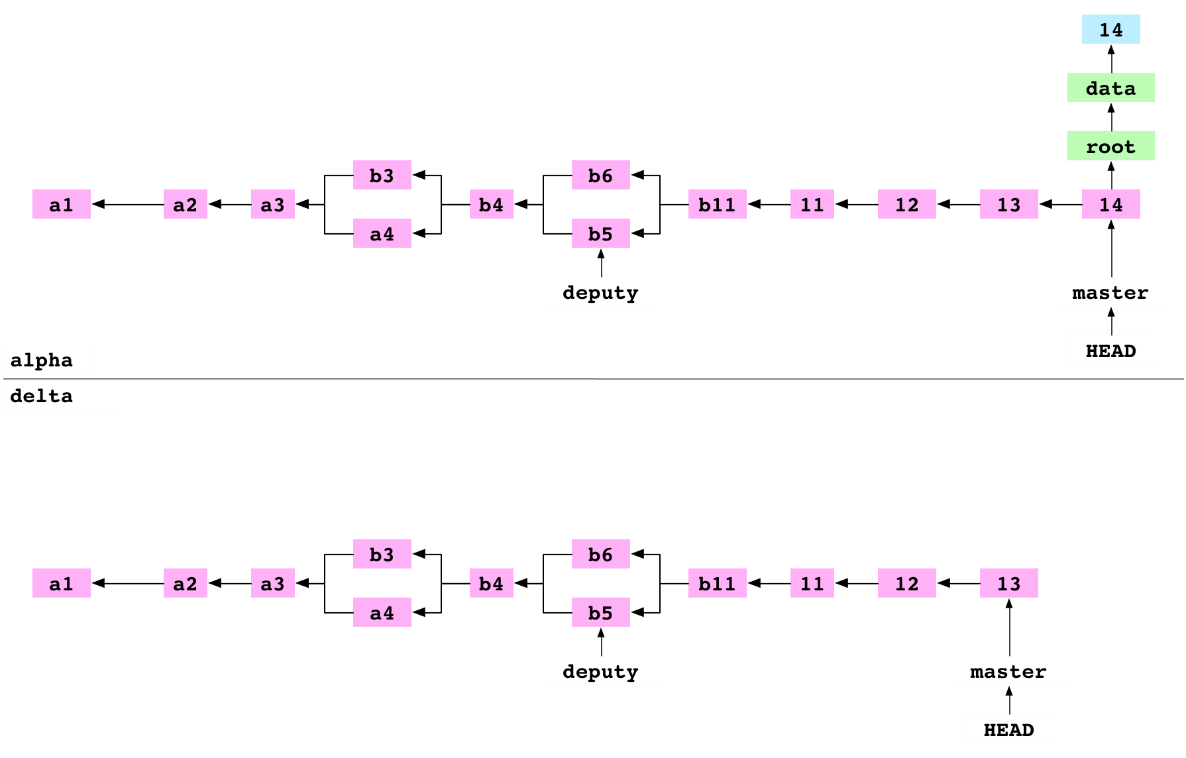
~/alpha $ git remote add delta ../delta

They set the content of data/number.txt to 14 and commit the change to master on alpha.

~/alpha $ printf '14' > data/number.txt

~/alpha $ git add data/number.txt

~/alpha $ git commit -m '14'



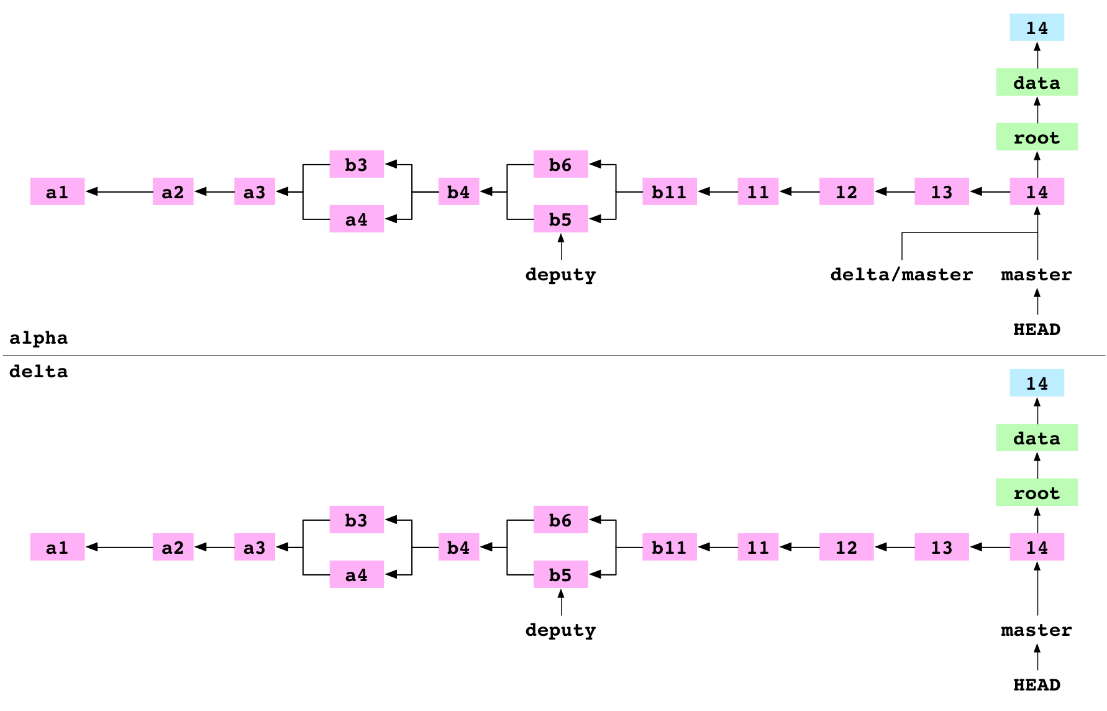
git push delta master

They push master to delta. Pushing has three steps.

First, all the objects required for the 14 commit on the master branch are copied from alpha/.git/objects/ to delta/objects/.

Second, delta/refs/heads/master is updated to point at the 14 commit.

Third, alpha/.git/refs/remotes/delta/master is set to point at the 14 commit. alpha has an up-to-date record of the state of delta.



***Git Object Types***

Git objects are the actual data of Git, the main thing that the repository is made up of.

All of these types of objects are stored in the Git Object Database, which is kept in the Git Directory. Each object is compressed (with Zlib) and referenced by the SHA-1 value of its contents plus a small header (SHA stands for Secure Hash Algorithm)

In Git, the contents of files are stored as blobs. It is important to note that it is the contents that are stored, not the files. The names and modes of the files are not stored with the blob, just the contents.

Directories in Git basically correspond to **trees**. A tree is a simple list of trees and blobs that the tree contains, along with the names and modes of those trees and blobs.

**The Commit**

The commit is very simple, much like the tree. It simply points to a tree and keeps an *author*, *committer*, *message* and any *parent* com­mits that directly preceded it.

The Tag

The final type of object you will find in a Git database is the **tag**. This is an object that provides a permanent shorthand name for a par­ticular commit.



The Git Data Model

the Git object data is a *directed acyclic graph.*

The cheap references I’ve represented as the grey boxes, the immutable objects are the colored round cornered boxes.



References

In addition to the Git objects, which are immutable – that is, they cannot ever be changed, there are references also stored in Git. Unlike the objects, references can constantly change. They are simple pointers to a particular commit, something like a tag, but eas­ily moveable.

A branch in Git is nothing more than a file in the .git/refs/heads/ directory that con­tains the SHA-1 of the most recent commit of that branch

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

How does Git actually retrieve these objects in practice?

Well, it gets the initial SHA-1 of the starting commit object by looking in the .git/refs directory for the branch, tag or remote you specify. Then it tra­verses the objects by walking the trees one by one, checking out the blobs under the names listed.

In fact, in Git the act of creating a new branch is simply writing a file in the .git/refs/heads directory that has the SHA-1 of the last commit for that branch.

Switching to that branch simply means having Git make your work­ing directory look like the tree that SHA-1 points to and updating the HEAD file so each commit from that point on moves that branch pointer forward (

***The Treeish***

Besides branch heads, there are a number of shorthand ways to refer to particular objects in the Git data store. These are often referred to as a *treeish*. Any Git command that takes an object – be it a commit, tree or blob – as an argument can take one of these shorthand versions as well.

* Full SHA-1

dae86e1950b1277e545cee180551750029cfe735

* PARTIAL SHA-1

dae86e

the full SHA-1 can be referenced fine with the first 6 or 7 characters. Git is smart enough to figure out a partial SHA-1 as long as it’s unique.

* Branch or tag name

Anything in *.git/refs/heads* or *.git/refs/tags* can be used to refer to the commit it points to.

* date spec

master@{yesterday}

master@{1 month ago}

* ordinal spec

master@{5}

This indicates the 5th prior value of the master branch. Like the *Date Spec*, this depends on special files in the *.git/log* directory that are written during commits, and is specific to *your* repository

* Carrot parent

dae86e^N

this refers to the Nth parent of that commit. Only really helpful for commits that merged two or more commits

* Tilde spec

dae86e~N

refers to the Nth generation grandparent of that commit

dae86e~5 ⬄ dae86e^^^^^

* tree pointer

e65s46^{tree}

This points to the tree of that commit. Any time you add a ^{tree} to any commit-ish, it resolves to its tree.



* Tree pointer

dae86e^{tree}

This points to th tree of that commit

blob spec

master:/path/to/file

This is very helpful for referring to a blob under a particular commit or tree.

**Git repository**

When you initialize a Git repository, either by cloning an existing one or creating a new one, the first thing Git does is create a Git directory. This is the directory that stores all the object data, tags, branches, hooks and more. Everything that Git permanently stores goes in this single directory. When you clone someone else’s reposi­tory, it basically just copies the contents of this directory to your computer.

When you run git init to initialize your repository, the Git directory is by default installed in the directory you are currently in as .git. The Git directory for our little project looks something like this:

For now, let’s go over some of the more important contents of this directory.

* .git/config

This is the main Git configuration file. It keeps your project specific Git options, such as your remotes, push configurations, tracking branches and more.

* .git/index

This is the default location of the index file for your Git project.

* .git/objects/

This is the main directory that holds the data of your Git objects – that is, all the contents of the files you have ever checked in, plus your commit, tree and tag objects.

The files are stored by their SHA-1 values. The first two characters make up the subdirectory and the last 38 is the filename

* .git/refs/

This directory normally has three subdirectories in it – *heads*, *remotes* and *tags*. Each of these directories will hold files that correspond to your local branches, remote branches and tags, respectively

* ..git/HEAD

This file holds a reference to the branch you are currently on. This tells Git what to use as the parent of your next commit

* .git/hooks

Contains shell scripts that are invoked after the git command

**Working directory**

Your working directory is temporary – everything is stored permanently in your git repository. Your working directory is a just a copy of a tree so you can edit it and commit changes

The Index

The index was called the cache for a while, because that’s largely what it does. It is a staging area for changes that are made to files or trees that are not committed to your repository yet. It acts as sort of a middle ground between your working directory and your repository. When you run git commit, the resulting tree and commit object will be built based on the contents of the index.

Now that you *hopefully* understand what Git is designed to do at a fundamental level – how it tracks and stores content, how it stores branches and merges and tracks remote copies of the repository,