A. CHRISTOPHER TOREK

DISTRIBUTED VERSION CONTROL WITH GIT AND MERCURIAL

copyright stuff

Dedication

NB: this front matter is still quite a hodgepodge; these are just various thoughts.

[Find out if I can use a copy of xkcd #1597]

Git is notoriously difficult for beginners. In xkcd comic #1597, Randall Munroe, referring to Git, draws a character (called "Cueball" on the explainxkcd site) who says:

> Just memorize these shell commands and type them to sync up. If you get errors, save your work elsewhere, delete the project, and download a fresh copy.

[Maybe turn the above into an epigraph?]

Everyone makes mistakes. The difference between being a novice and being an expert cannot be boiled down to just one sentence, but we can say that one—maybe the most important—difference is that an expert can recover from mistakes mid-process. This book should help you do so.

Git makes it easy to make mistakes, and also easy to correct them. Mercurial makes it harder to make mistakes, but also harder to correct them.

There are already several good Git books, Chacon and Straub [2014] and Loeliger [2009]. The primary author of Mercurial has published a Mercurial book, O'Sullivan [2009]. So why write another book?

Loeliger's book is good but has become out of date—a constant hazard with actively-developed software. Chacon's book is on line and gets updated, but focuses strictly on Git. O'Sullivan's book focuses strictly on Mercurial. I'm not currently aware of any books that approach version control in this particular manner, and show both Git and Mercurial usage.

If you are reading this book, you probably have thought about using Git or Mercurial (or even both), or you may have used them in the past or be using them now and want to learn more. You may be considering which one to use. The book will try to address all of these.

I also wanted to have a book that could also appear as a series of web pages that were structured very differently. That is, the book would proceed in a logical building-up fashion, but using a web hyperlink, you could start with any particular topic and zoom up or down the scale of generalization or specialization to find specific answers.

Part of the project was to write a program to produce hypertext (HTML) web pages. It would read the LaTeX input, and actually *use* LaTeX to generate figures, but keep the small-section-at-a-time setup.

The text for the book and the text for the hypertext setup would live together in (at least relative) harmony. At the time I write this, the outcome of this experiment has yet to be determined.

Both Git and Mercurial have many Graphical User Interfaces (GUIs) and Integrated Development Environments (IDEs) that allow you to browse commits, and in some GUIs and all IDEs, change or create branches, make new commits, and so on. Every one of these is different and we cannot possibly address them, so we will stick with the command line interfaces.

[Here's an intro bit that goes with the xkcd comic]

[scene: you've been given some shell commands to type]

```
$ git clone ssh://host.name/path/to/repo
$ cd repo
make changes...
$ git commit
repeat change-and-commit as needed
$ git pull --rebase
```

You may get a merge conflict, and not know what to do. Or maybe you do know what to do, and have done it. But in any case, you want to continue and you try (as instructed):

```
$ git rebase --continue
```

but now you get an error:

```
No changes - did you forget to use 'git add'?
If there is nothing left to stage, chances are that something else
already introduced the same changes; you might want to skip this patch.

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To check out the original branch and stop rebasing run "git rebase --abort".
```

If you run `git status`, which is a good thing to use, you simply see:

```
# Not currently on any branch.
nothing to commit (working directory clean)
```

You may also run into problems when you have used `git merge` or `git rebase` successfully—or so you thought; and then you discover that you want or need to back out of the merge or rebase.

This book should set you up so that you know what to do.

Organization of this book

The book begins with an overview of version control in general. We introduce terminology that you will need, and review some historical version control systems and their distinguishing characteristics.

Next, we cover graph theory and how it applies to both Git and Mercurial. It's worth noting that while this theory has nothing to do with the controlled source itself, it's a basic building block for *performing* source control. It not only interacts with merging and rebasing, it is also fundamental to the distributed nature of Git and Mercurial repositories.

The third chapter describes more precisely what is in a commit; how we compare one commit to another; and how, at a high level, merging works, using the commit graph described in Chapter 2. It also mentions the issues with file path names that will affect you once you distribute a repository across dissimilar operating systems.

The fourth chapter covers the mechanics of distributing repositories, and one of the key consequences: that some commits are *public* and some commits are *private*. Private commits can be deleted without affecting others, but once a commit is published, it may be impossible to retract it. It also includes some of the theory needed to understand how commits can be signed and authenticated.

With these basics out of the way, Chapters 5 and 6 some of the basic setup and usage of both Git and Mercurial. We discover just how similar, and in some cases just how different, the two VCSes are.

XXX this is now wrong Chapter 6 discusses *diffs*: comparisons between pairs of commits, or one commit and the corresponding working-tree files.

Chapter 7 covers merging, which—while it has many variations—basically amounts to combining two diffs.

Chapter 8 (. . . is not yet written).

Those who want to jump right to using Git or Mercurial can start at Chapter 5, referring back to earlier theory chapters only as needed. However, careful reading of the history and theory chapters should give you a much better idea of what you are doing with the practical aspects of version control.

Each page has room for graphics, side notes, and exercises. Side notes that are numbered are specific details regarding items in the main text. Unnumbered side notes are general ideas I find interesting or relevant, yet not directly related to the main text. The exercises are optional, but are meant to verify and cement your understanding of the concepts involved.

ASCII

Chapter 3 refers to ASCII, the American Standard Code for Information Interchange. This is a very old standard for saving and exchanging data on computer systems, in which one single-byte code represents one letter, digit, or other printable symbol (and certain

"control" operations including tab, carriage return, and the like). ASCII is an old standard, and by the 1980s, all computers could work with it. It is not adequate to modern needs, but much is built upon it.

Numbers

This book mostly works with ordinary decimal numbers. However, hashes are typically encoded in hexadecimal, with "digits" that range from `0` through `9` but then continue on with `abcdef`, which may be written in either uppercase or lowercase. In some places, we will use a leading zero and letter-x to denote hexadecimal numbers: `0x10` represents the same number as 16, `0x80` represents the same number as 128, `0x100` represents the same number as 256, and so on. We will write hashes as `a27fc31` and the like, without any leading prefix. While these do represent numbers inside the computer, their decimalized equivalent representations are not useful for anything.

Bugs

The term *bug* dates back to at least the 1870s and Thomas Edison. The first application to computing may have been in 1947 when Grace Hopper's group at Harvard discovered a moth in the circuitry of the Harvard Mark II computer. (The log book containing the remains of the moth is now in the possession of the Smithsonian Institution; see Smithsonian Institution [1994]). Bugs are, however, usually very small, difficult to observe from a distance, and can induce a great deal of revulsion in some people. In this book, we will instead use larger, friendlier mammals, specifically marsupials. Instead of moths, ants, spiders, centipedes, and cockroaches, we will deliberately introduce kangaroos and wallabies into our programs and processes, so as to illustrate their removal.

Target Audience, preface, introduction?

Animals in this book

The Marsupial Maker is not a real project, but marsupials are real creatures, and I find them quite interesting. Here are photos of some that I took on a trip to parts of Australia in February of 2010.

This stuff is currently at front of book, but probably should be at back of book.

Plate 1: Red kangaroo, Healesville Sanctuary.

The kangaroo is probably the most widely known marsupial. There are actually four species of large kangaroo: the red, the eastern and western grey, and the antilopine. There are also smaller tree-kangaroos and rat-kangaroos.

Plate 2: Bennet's wallaby, Cradle Mountain.

The wallaby is smaller than any kangaroo, but in fact the term "wallaby" is defined a bit loosely. There are many varieties of this creature and they all resemble kangaroos, both in overall shape and in their hopping gaits. This is a red-necked or Bennet's wallaby.

Plate 3: Pademelon, Cradle Mountain.

The pademelon is even smaller than the wallaby. They are common in Tasmania. I think the smaller one is probably a nearly full grown joey although they could be male (larger) and female (smaller).

Plate 4: Koala, Healesville Sanctuary

The koala vies with the kangaroo for being most widely known marsupial. Koalas mostly eat eucalyptus leaves, which are not very nutritious and actually rather toxic.[1] As a result they spend most of their time sleeping. Although they are very cute, they are not very bright and can be rather aggressive. If you would nonetheless like to hold a koala, note that this is permitted only in the states of South Australia and Queensland. There are koala sanctuaries in Queensland, in Brisbane and near Cairns. (Cairns is also a good base for a visit to the Great Barrier Reef.)

[1] Eucalyptus trees transplanted to California use their leaves to kill off native trees. The sticky oil from the leaves is fragrant but unpleasant to discover on your car, and the trees contributed to the 1991 Oakland firestorm.

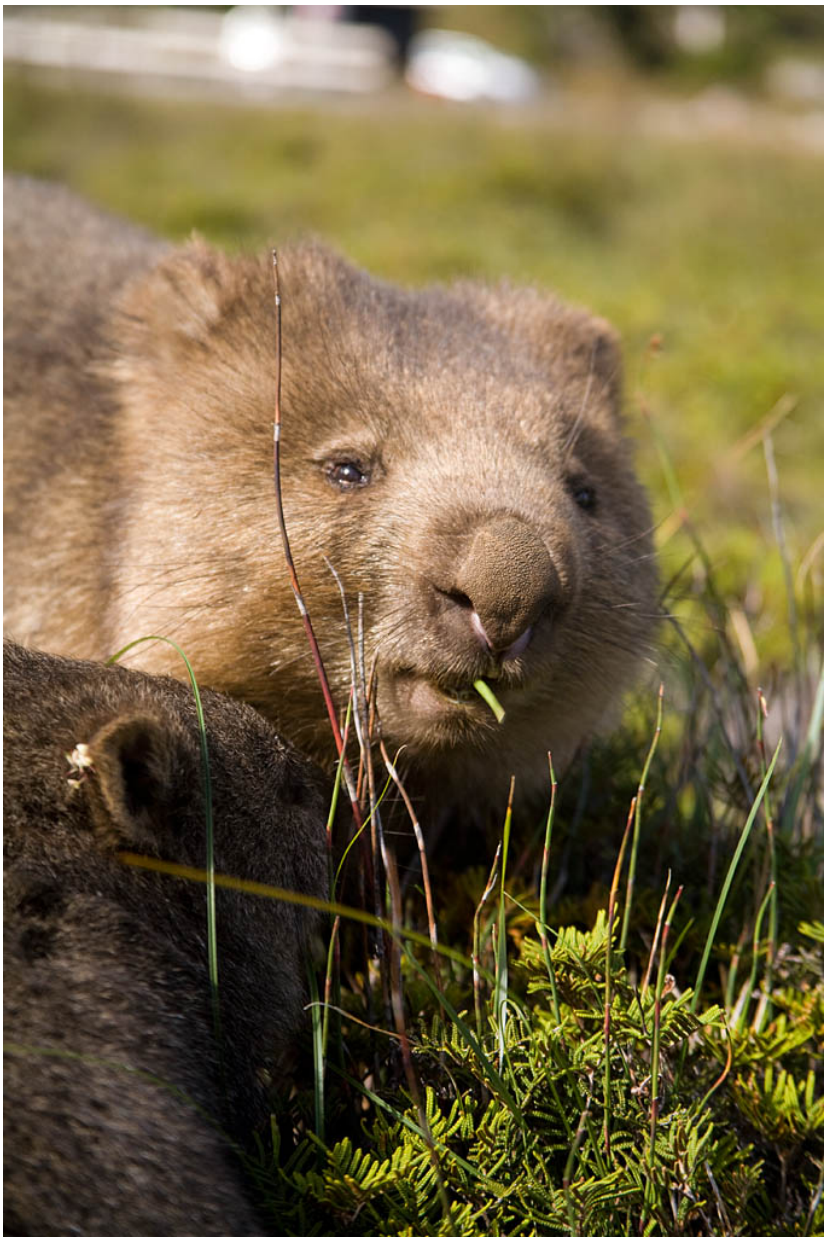

Plate 5: Common Wombats, Cradle Mountain.

Wombats are the gophers of Australia: they dig tunnel systems. They look like big sleepy slow-movers but they can move very fast when it suits them. Despite the name Common Wombat, these wombats (and the other two living species, the northern and southern hairy-nosed wombats) are not very common, having been hunted and treated as pests for decades.

Plate 6: Tasmanian devils, Cradle Mountain.

Tasmanian devils look nothing like the cartoon version. They got their name from the loud and rather scary noises they make when feeding. In fact, though, while they fight with each other, they generally try to avoid humans. They are crepuscular[2] and nocturnal eaters, mostly of carrion, a lot of which is roadkill. This unfortunately puts the Tasmanian devils in the position of becoming roadkill themselves.

They have the fascinating misfortune to be susceptible to a bite-transmissible cancer called *devil facial tumour disease*. Although at first there was a theory that there was some oncovirus[3] involved, the current best idea is that a genetic bottleneck has given most Tasmanian devils an insufficient variety of Major Histocompatibility Complex variants.

[2] Dusk and dawn.

[3] Cancer causing virus.

Plate 7: Spotted-tail quoll, Cradle Mountain. I apologize for the somewhat low quality of this image. My flash batteries went flat at this point, and the quoll was restless and difficult to photograph.

The spotted-tail quoll (also known as the spotted quoll and tiger quoll) is nocturnal and more common than its Tasmanian devil relative. The quoll prefers a wetter climate, and is found wild in mainland Australia as well as on Tasmania. They are generally much quieter than devils, although they can make a screaming noise that has been compared to the sound of a circular saw. They eat various insects, reptiles, and birds, but they will also eat small or injured wallabies and pademelons, and will consume carrion.

Plate 8: Numbat, Perth Zoo.

Numbats are the quaggas of Australia: impossible to believe in even after you have seen one.[4] The numbat is also in danger of sharing the quagga's extinction,[5] with an estimated total population of under 1000 individuals (I found this number in several 2008 and 2010 reports but was unable to find anything newer).

The numbat is the emblem mammal of Western Australia. These small, diurnal marsupials eat termites, and one numbat can eat 20,000 termites a day. The Perth Zoo is involved in a captive breeding program with releases to the wild. I hope these little guys recover.

[4] This award should really go to the duck-billed platypus, but the platypus is a monotreme rather than a marsupial. British naturalist George Shaw suspected a hoax when he received a specimen in 1799, as did several more naturalists over the next few years.

[5] The original African quagga, a sub-species of zebra, is now extinct. The Quagga Project is attempting to restore a similar sub-species by selective breeding, with some success so far; see Heywood [2013].

Contents

1
Version Control: Concepts and History

A customer comes to you and says that your Marsupial Maker is producing wallabies instead of kangaroos. He can't or won't (for size or data sensitivity reasons) show you his *entire* input, but he has a stripped down example for you. When he gives you his simplified example, you try it and it works fine. Apparently it works in your latest version, but not in his. You must have fixed the problem already! You could just tell him to upgrade, but if he does upgrade and the problem persists, you have wasted his time and annoyed a customer.

To prevent him from becoming an ex-customer, it would be nice if you could go back to your old version and run his sample and observe the problem. Then you could see what you've changed that might have either papered over the problem, or actually fixed it. This won't *guarantee* that your latest version is good, or that you will find the true root-cause of the problem, but it certainly improves your chances.

What is a version control system?

At its core, a *version control system* provides the ability to choose between older and newer versions of data as stored in computer files. Entering these files into a version control system (VCS) is called *versioning* or *version-controlling* the files. The file contents are typically source code for computer programs, but any computerized files can be version-controlled. For simplicity, we'll refer to this as *source code* that is made up of *source files*, but keep in mind that you can version-control things that are not particularly source-like. The document files for this book are version-controlled, for instance.

Simply storing a new copy of a file every time it is changed—some file systems provide this directly[1]—could be viewed as a basic form of version control, but these are perhaps best thought of as degenerate cases. Minimal version control systems add features such as date- and name-based retrieval. You could ask the system to let

[1] For instance, Files-11 on VMS and the TENEX file system offer this option, and various write-once (e.g., CD or DVD-ROM) file systems *must* do it.

you retrieve everything the way it was one month ago, or at a time when you marked the source with the label "version 0.1-alpha", for instance. They also provide ways to view and compare versions, to answer questions you might have such as "what, in general, has happened to the source over time," or to show changes in a particular file or set of files between specific marked versions.

Thus, a VCS acts as a database of source files, with some way of retrieving specific versions of the files. This database is called the *repository*. When you add updated versions of your source files, the database stores the updated sources inside the repository. Each version of a file is also called a *revision*, so that if you fix a spelling error and enter the updated file into the repository, you now have another revision of that file.

Used as a verb, *to version* means *to put under control of the VCS*. Used as a noun, *version* means *a specific version taken from the VCS* (of one file, or of a group of files). Usually the noun form appears with additional modifiers, as in the phrase *the old version of* `kanga.c` or *version 2.1 of* `roo.c`. If no specific files are listed, we typically mean everything, or at least everything recently under discussion: *version 2.1* (of everything in the repository, or of the specific files and/or directories we were talking about). The word *revision* is always a noun, but otherwise means the same thing as *version*.

Another verb, *to check in*, means *to store into the VCS*. As you might expect, if we can *check in*, we can also *check out*, meaning *extract from the VCS*. Some VCSes add the verb *to update*, which they may use to distinguish between extracting an older version (which you *check out*) and moving up to the latest and (we hope) greatest (to which you *update*). Mercurial uses *update* as a pure synonym for *checkout*.

Check in is sometimes hyphenated (*check-in*), or written as one word, *checkin*. These avoid ambiguity: *I'll check in* `roo.c` (look through the code to see if any wallabies got in there) vs. *I'll check-in* `roo.c` (to store a new revision). *Check out* is likewise often written as one word, *checkout*, or hyphenated, but the verb form is still *checking out*, which reads much better than *checkouting*.

Newer VCSes add more noun-and-verb words:

- *To commit* means much the same as *to check-in*, but with some technical differences we will see in a moment. As a noun, *a commit* also refers to a version, but specifically one as stored by the verb form of *commit*.

- *To clone* is basically a fancy term for copying an entire repository, often from a different machine over a computer network, e.g., from a web site. As a noun, *a clone* is a repostory made by cloning.

- *To fork* is functionally the same thing as cloning, but usually with

a different intent. The noun form *a fork* is thus the same as a clone, but those making making a fork may intend for their work to diverge with little or no re-synchronization (perhaps forever or perhaps for a limited time), and/or for still others to collaborate via their fork instead of the original.

Why have version control?

With simple projects, you can handle version control by making regular backups or archives.[2] VCSes generally also provide *metadata*,[3] such as log messages and automated date-and-time stamping. VCSes commonly use their data and metadata to provide the following:

Versioning and repeatability This is the most basic part of version control. Any VCS *must* provide the ability to retrieve a previous version, the way it was stored at the time it was stored.

Accountability and tracing If you're the sole developer on a project, this is not really useful: everything that ever changed is your own doing, whether the change was good, bad, or incomprehensible. If you're on a team or collaborating, though, knowing *who* made some change in the past may be just as important as finding the actual change. If nothing else, this allows you to ask the original author about intent: *why* did she make that change?

Customized fixes Sometimes customers are unwilling or unable to upgrade. This happens often in the embedded systems world, where safety-critical systems like airplane controls, medical products, and so on were tested and certified: upgrading can require a full re-test and re-certification. Bug fixes to particular sub-components may be allowed with fewer expensive and time-consuming tests.

In this case, having found the root cause of a particular customer's problem tied to a specific version of your product, you can incorporate a specfic fix and deliver a customized variant of the product. (It's worth noting that this can become a headache of its own: you should weigh carefully the pros and cons of providing a customized variant of your product, as the customer may demand additional future changes done the same way. However, a good version control system can make this process much simpler than it would otherwise be.)

Simplifying development Whether you're controlling application software, documentation, web page themes, recipes for actual edible cookies, or any other files on a computer, you will often find yourself doing iterative and incremental development. Here, you provide new versions with refinement and/or additional features and

[2] In fact, there is a fairly close relationship between system backups and version control. The key difference between the two is a function of their *purpose*.

Backups aim to allow you to restore anything—one file, many files, or even the entire system—after some kind of error or disaster, including loss of storage media. Backups therefore tend to be performed on a time schedule, such as hourly, daily, weekly, and so on. When backups are aimed at disaster recovery, we may delete intermediate versions, e.g., discard all hourly backups after a daily backup, then discard all daily backups after a weekly backup. In other words, backups are usually made with a system-driven point of view.

Version control systems, by contrast, aim to allow you to view or restore files from a user- and/or project-driven point of view. New versions are entered at check-in or commit time. We'll see more about this below. If discarding old versions is allowed at all, it is also normally done with specific care, rather than according to a time schedule.

If you make backups at well-chosen times, and keep those backups forever, this *does* result in a form of version control. Management and comparisons of versions may prove difficult, though.

[3] Metadata is simply data about data. In this case, it's information about the versioning process.

Incremental development has a long history, going back to the 1950s; see Larman and Basili [2003].

obtain user feedback as you work. Version control allows you to step back to any previous version if the latest changes are disapproved, or to find where bugs have crept in over time.

A good version system also allows parallel development of *different* features (ideally completely unrelated, though often the ideal is impossible, and in some cases the features may even be deeply intertwined). These features can be produced as independently as feasible, then merged back into the main-line development, and/or into each other, using the version control system's tools. Isolating each feature allows you to focus on one thing at a time, to whatever extent is allowed by the problem itself. Moreover, if you make many small, incremental steps, then discover a problem as you approach the finished feature, you may be able to re-use most of your work.

dump the parenthetical?

Integration with bug-tracking systems Version control systems can use file data or log message metadata to associate particular fixes with particular bugs.

Automated testing Using commit atomicity (which we'll define in a moment), and optionally tied together with bug-tracking systems, a version control system can automate testing, either at the time the change is made, or after the fact for finding regressions.

Centralized vs distributed

Many older VCSes are *centralized*, or CVCSes. Git and Mercurial are DVCSes: *distributed* version control systems.

The key difference between these two kinds of systems is that a centralized VCS has a designated master repository. There may be multiple copies of the master, or even multiple masters with some kind of synchronization protocol (e.g., ClearCase MultiSite), but there is only one master. Their design *assumes* this single-master-ship and thus is allowed to depend on it.

With a distributed VCS, there is no designated master repository. Users generally have a complete, private copy of each repository. Communications between these private copies are, at least in principle, peer-to-peer operations: neither repository is any more masterful, and conflicts—situations where both Alice and Bob have made changes to the same regions of the same files—can and do occur and require some kind of resolution.

It's always possible to use a distributed VCS in a centralized manner: you simply designate one particular repository as the master version, and coordinate updates to it. However, centralized systems often provide features like locking source files or directories, re-

stricting access (for read and/or write, to particular files, directories, and/or branches), and so on. With a typical DVCS it's more difficult (though not technically impossible) to provide these, and Git and Mercurial simply don't, at least not without add-ons. With CVCSes that provide locking, users may lock files (typically just one specific version ID) to prevent other users from making conflicting changes. This is conceptually easier, but of course it can prohibit parallel work.

In my opinion, most of these features fill a much-needed gap.

A side note on trees

The word *tree* is rather heavily overloaded in this book, and in computing in general. Below, we talk of *work-trees*, which use the underlying operating system's file-system trees consisting of directories—sometimes called folders—that may contain files (which are still always called files) and additional directories, which in turn may contain still more files and directories.

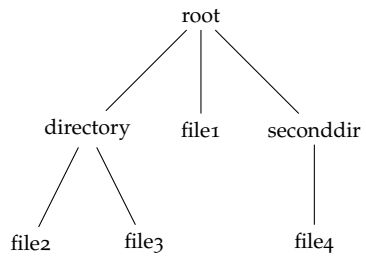

Figure 1.1: Tree of files in a file system.

Computer scientists prefer to draw their trees upside down, with the root at the top and branches growing downward, as in Figure 1.1. We'll also see cases like Figure 1.4 where we draw our trees sideways, with the root at the left.

In the future, tree-structured file systems may well seem quaint. Today, attaching attributes to files (e.g., tagging email, photographs, and StackOverflow postings) is augmenting or even replacing tree-oriented lookup. For now, though, the versioning systems still use trees.

Besides storing trees made up of files and directories, we find that version control systems must implement their own version trees. For instance, suppose you store into the repository a change to one particular file, such as `kanga.c`. Then you use the VCS to go back to the previous version of `kanga.c`. While still using that previous version, you store a different change, You have now created a version branch. The old `kanga.c` now has two new revisions. Both have the same *parent version*, so they are siblings in their family tree, as it were. Each of these sibling versions can act as a parent to another version of `kanga.c`. The VCS must be able to compare any revision to its parent, so it needs to build the kind of tree shown in Figure 1.2. Just as real trees have branches and roots, these version trees also have branches and roots. The word *branch* can mean a branch in this version tree; we'll see more precise definitions later. (Also, you might wonder—in fact, you *should* wonder—how we can distinguish between all these `kanga.c` files when they are arranged in a version tree like this. We'll see more about this soon.)

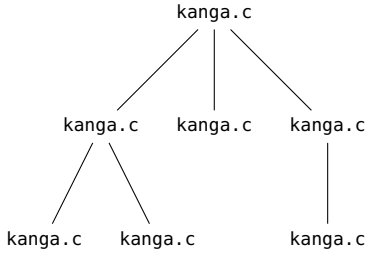

Figure 1.2: Tree of versions of one file.

Repositories and work-trees

VCSes distinguish between the repository (where files are well-controlled and versioned) and the *work-tree*[4] (where files are usually not versioned). The work-tree is typically where you edit the files, compile them, and otherwise work with them. We already noted

[4] The terms work-tree, working tree, and work directory, hyphenated or not, are all used interchangeably.

these verbs, but now we can describe them in more detail: *checking out* or *updating* extracts a version from the repository to the work-tree, and *checking in* or *committing* stores a new version from the work-tree into the repository.

With a centralized VCS, the master repository can be left on a centralized server. We can then checkout to a work-tree on the user's machine (e.g., a laptop) without first copying the entire repository, so the laptop's storage can be smaller than the server's. Typically we can also extract only a small subset: if the repository contains hundreds of packages, libraries, or other subsystems, we can check out just one subsystem, or even just one file. This is convenient when one is just making a quick and easy change. On the other hand, it requires that the work-tree be connected (networked) to the centralized server during checkout and checkin/commit operations, and if the local workspace is disconnected, other revisions may not be available.

Since distributed VCSes usually copy the entire repository,[5] the entire history is normally available at all times. The main tradeoff here is longer setup times for the initial copy (the *clone* operation), and additional non-volatile storage needed for the clone. These DVCSes work hard to make synchronization operations efficient, so that once you have the initial clone, obtaining new versions is relatively fast. (For instance, I have seen initial clones that take four or more hours over slow networks, but their resynchronizations usually take only a few seconds.)

[5] Both Git and Mercurial now support *shallow clones* and/or *single-branch clones*, which can omit some of a repository. We will address these later.

Atomicity: what is the smallest unit of revision?

Older VCSes work with just one file at a time, using the check-out / check-in model. Their unit of atomicity is the *file*. Even if you check out (or in) many files at once, the VCS just does each operation on a per-file basis, as if you had done them one at a time. Consider the four buildable iterations shown in Table 1.1. Let's assume that at each iteration, a new set of compile-able files were all checked in together—but our VCS only works with files, one file at a time. Every file starts out as version 1, but at iteration 3, file `kanga.c` has two versions, while file `roo.c` has three.[6] The last buildable iteration introduces the new file `wallaby.c`, which is now at version 1. Which versions of which files do you need in order to build any given iteration? Which file-version combinations do you need to skip? The answer is in our table, of course, but the VCS does not track this on its own.

I mention three historical version control systems by their acronymic names below: SCCS, RCS, and CVS. See Table 1.3 for what these acronyms stand for.

[6] For the moment, we will just number each file revision, without worrying about making trees out of the revisions.

Newer systems, including Git and Mercurial, work on larger sets of files. Their unit of atomicity is the *commit*. Committing a change enters all the files at once. If anything goes wrong, *no* files get a new

check-in	iteration	files		
1		`kanga.c:1*`		
2	1	`kanga.c:1`	`roo.c:1*`	
3		`kanga.c:2*`	`roo.c:1`	
4	2	`kanga.c:2`	`roo.c:2*`	
5	3	`kanga.c:2`	`roo.c:3*`	
6		`kanga.c:3*`	`roo.c:3`	
7	4	`kanga.c:3`	`roo.c:3`	`wallaby.c:1*`

Table 1.1: Four buildable iterations, recorded with file atomicity, resulting in seven check-ins. The file actually checked-in on each row is marked with an asterisk.

revision; if the entire commit succeeds, *all* files get a new revision, as shown in Table 1.2. Extracting the latest commit—row 4—gets you the latest version of all three files. Backing up one version gets you the previous `kanga.c` and `roo.c`—this changes the *contents* of `kanga.c` while keeping the *contents* of `roo.c` the same—and removes `wallaby.c` entirely, all automatically.

commit	files		
1	`1:kanga.c`	`1:roo.c`	
2	`2:kanga.c`	`2:roo.c`	
3	`3:kanga.c`	`3:roo.c`	
4	`4:kanga.c`	`4:roo.c`	`4:wallaby.c`

Table 1.2: The same four buildable iterations, but with commit atomicity.

Generally, in file-atomicity systems you can name or *tag* a set of file-revisions that go together, and extract by tag. Tags tend to have a noticeable cost—even if they don't use a lot of space or time,[7] they present a sort of revision clutter, and in practice they're used only for more-major checkpoints. Commit-based systems obviate the need for these tags (though as we will see, tags are still useful).

[7] Tags in CVS, for instance, are maintained on a per-file level, so that tagging an entire tree is a very slow operation.

A system with commit-based atomicity could still store individual files labeled with their own individual revisions in its internal repository structure. In other words, the system may simply keep its own tables mapping from commit to file revisions. The system may also have you check-in or add individual files, then commit the changes as a whole. Ideally, whatever internal method the VCS uses is invisible, but in practice, some of the seams may show.

Exercise 1.1: Suppose you were building a commit-based VCS using some existing file-based VCS to do the file-storage. How might you take a request of the form "give me commit 3 in my work-tree" and turn it into check-outs of the proper file versions? Do you need Table 1.2 for this?

Compression

One natural objection to keeping every version of every file is that this will require too much storage space. VCSes therefore often use file compression techniques. Ordinary compression algorithms such as Huffman encoding, Lempel-Ziv, and so on are useful here: for instance, Git uses zlib's Deflate algorithm. However, given the nature

of version control and the desire to be able to view the differences between different versions of files, it makes a great deal of sense for VCSes to use *delta compression*.[8]

Consider what happens with a single source file when you commit a change. Suppose, for instance, that you replaced one line of code with another different line, added a comment line, and removed an unused variable. Regardless of how long the original source is, if we already have the previous version of the file, we can save the new version by saving only *instructions*, saying how to modify the previous version to produce the new one. In this case, the instructions would read: delete the replaced line, insert the new version of the replaced line, insert the new comment line, and delete the removed variable.

The technique described above is a *forward delta*, which converts an older revision into a newer revision. Many VCSes that use deltas use *reverse deltas*, storing the latest variant intact and providing instructions for moving back in time to older versions.[9] This makes sense since we tend to work on the latest code more often than on older versions, and it's faster to extract the latest version intact, with the time needed to get an older version being proportional to the number of deltas to apply. At the same time, though, reverse deltas present implementation issues in branch-y revision structures.[10]

Note that commiting a file with no changes to it results in perfect delta compression: the instructions are "make no changes", i.e., the instruction list is empty. This means that in practice, commit-based systems use no more storage than file-based systems, even though every commit must save every file every time.[11]

Mercurial uses forward deltas internally with a simple scheme to avoid having to chase long delta chains: when the chain is getting too long, store a new full (but still zlib-deflated) copy. In any case its implementation details are so well hidden as to be completely invisible normally.

Git is often said not to use delta compression, which is true on one level, but not on another. Like Mercurial, Git sets limits on delta chain lengths. Git hides this compression in its *pack files*, using a very clever and very unusual scheme. Its implementation is properly abstracted away, so that the main place that its delta compression shows through occurs when you see its *Delta compression using up to* n *threads* progress messages.

[8] Delta compression is a specific form of the generalized *string to string edit problem*. We want to find a *minimal edit distance*, i.e., the fewest changes needed to transform one string into another. We use just two instructions: delete and insert (sometimes we see a third instruction, replace, but replace is simply shorthand for delete-then-insert). Allowing additional operations such as moving substrings can produce much smaller edit distances, but the time complexity required to find them increases. See, e.g., Cormode and Muthukrishnan [2007].

[9] SCCS uses *interleaved deltas*, where extracting any version takes approximately linear time.

[10] For instance, RCS uses reverse deltas in what it calls its "trunk", but forward deltas within branches. See the sections below for RCS's trunk-vs-branch distinction.

[11] Of course, there are many more tricks commit-based systems can use, even if they don't use delta compression.

File identity

The *identity* of a file seems obvious: it's just a path name like `kanga.c` or `lib/marsupial.h`. However, over time, we find that files are re-

named, copied, deleted, and re-created. For instance, `lib/marsupial.h` might be named `include/marsupial.h` in earlier or later versions. Traditional VCSes need some way to track name-changes.[12] Often, path names or path name changes are stored as separate metadata, and the VCS turns the file name into an internal identifier (an object or inode number, for instance) so that the system can see that two different path names in two different revisions really refer to the same file.

[12] SCCS and RCS did not even attempt it: the name of the version-database file was determined by the name of the file within the work-tree, and vice versa. This method is not really acceptable today.

Normally, this automatic name to ID mapping goes smoothly enough, although you may need to inform the versioning system of name changes (e.g., using `hg mv` rather than plain `mv` in Mercurial).[13] However, removing a file and then trying to re-add a file with the same name (with or without a shared history) results in what are called *evil twins*: two identical pathnames that refer to different internal objects. These cause (VCS-specific) issues during merges. Git sidesteps this problem entirely using a unique strategy we will cover later, though it can still run into remarkably similar issues when working with both case-sensitive and case-insensitive file systems (e.g., Linux and Windows®): the user on the case-sensitive file system can create `roo.c` and `ROO.c`, which are different files, but the user on the case-insensitive file system cannot work with both files as the operating system insists both these names identify one single file.

[13] Mercurial is not actually doing name-to-ID mapping here. The `hg mv` step is instead recording directory modifications for the next changeset. To the user, though, this is a distinction without a difference.

Git does not handle this situation very well at all today. Similar problems can occur with pathname encoding, e.g., in UTF-8. We will see more about this in Chapter 3.

Branching and version numbering

Tables 1.1 and 1.2 simply number each revision sequentially, giving a simple linear model of development. Version control systems must provide richer models. They need not use numbers at all (and Git does not), but two important historical systems—RCS, the Revision Control System [Tichy, 1985], and SCCS, the Source Code Control System [Rochkind, 1975], do number each revision. Reviewing their numbering method is instructive, particularly in terms of the way they handle branching files.[14]

[14] These two systems use file atomicity, though the numbering method shown here would in principle work with commit atomicity.

RCS and SCCS start each file with a *pair* of version numbers, major and minor. Here the first version of `kanga.c` is not 1, but rather 1.1. By default, each check-in increments the second number, going to 1.2, 1.3, and so on.

We may choose to mark a check-in as major,[15] e.g., if we are making a new release of the Marsupial Maker. In this case, the VCS increments the first number and resets the second, giving us version 2.1, 2.2, and so on. When we release version 3, we can keep making improvements to the 1.x and 2.x versions, and when we release version 4, we can keep making improvements to all the old versions.

[15] This is not a technical term. We're just using it for now to separate the number before the period from the one after the period.

We can draw this as in Figure 1.3: a major number like 3 provides

the branch on which versions are committed; adding the minor number, to get 3.1 or 3.2, gives us the revision within the branch.

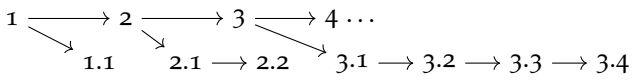

Figure 1.3: Straightforward two-part version numbering. You might wonder why we bother with the arrows here, since the numbers suffice. The answer is that we will soon remove the numbers. See Figure 1.6, for instance.

Assume Figure 1.3 shows all of the 2.x and 3.x versions of file `kanga.c`. It's easy to see the latest 3.x version is 3.4. However, our important Marsupial Maker customer is not using the latest version 3 release. We somehow[16] discover that he is using `kanga.c` version 3.2 and `roo.c` version 3.3.

We decide to produce a special fixed version for the customer. We track part of the problem to `kanga.c`. To fix this problem, we need to make a new sub-branch.

Extending our numbering system gives us an obvious way to number this particular sub-branch: starting from revision 3.2 of `kanga.c`, we make a 3.2.1 branch, and a new revision 3.2.1.1 within that branch. If we need two internal iterations to fix the problem, the second one will be 3.2.1.2, as in Figure 1.4.

[16] There were multiple schemes for this, including embedding per-file revision information directly in the product, or building manifests (lists) that map external releases to internal file-version-lists.

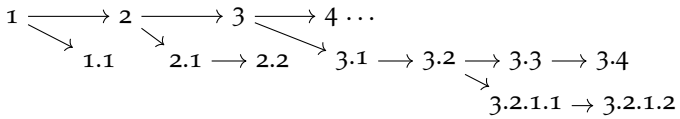

Figure 1.4: Numeric sub-branches.

Note that if, before we can produce the special fix for our important customer, we must fix `roo.c` as well, we need a new 3.3.1 branch in that file, as opposed to the 3.2.1 branch in `kanga.c`. This is a small issue on its own, but ultimately it proved very annoying, especially when this issue was multiplied by many thousands of files in large code bases.

This branch-and-revision numbering system makes a nice theory, but actual implementations sometimes get in the way. For instance, RCS's real revision structure involves adding two-component revision IDs to what it calls the *trunk*, and its branches begin only with three-part revision IDs (with revisions within those branches having four parts, sub-branches of branch-level revisions having five parts, sub-revisions having six parts, and so on). Meanwhile, the trunk can only be grown at its tip (the rightmost position in Figure 1.5). As a result, once revision 4.1 exists, we can no longer add a revision 3.5. Instead, RCS will automatically check in a new 3.x version (which should logically be 3.5) as new branch version 3.4.1.1, and the next one as 3.4.1.2, and so on. If we later wish to branch the original version 3.4 of that file, we start that branch with 3.4.2.1 rather than 3.4.1.1.

Exercise 1.2: Why do we add a *pair* of numbers for these sub-branches? Hint: consider what happens if a different, but also important, customer needs a different fix for `kanga.c` version 3.2, and that using the fix for the first customer complicates things. We'd like a new sub-branch of version 3.2; what series of numbers can we use?

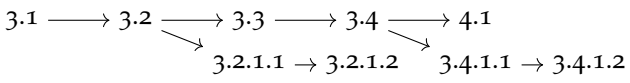

Figure 1.5: Actual branch structure in RCS. Versions before 3.1 are omitted to fit the diagram on the page.

The need to start a new branch to continue working with an earlier trunk version is also a relatively minor issue. However, like several other minor issues, this get multiplied across many thousands of files in a large system.

Commit-based systems remove the headache of having different numbers for each file: for the customer with the private marsupial fix, we need only find the identifier for the (single) commit from which the software was built.

In practice, many systems use multiple repositories so you may still need multiple identities. We'll see one method of dealing with this using subprojects, although these have their own drawbacks.

Branching with names

Numbering each of our branches (so that 3.2.1 is a branch of 3.2, and 3.4.1 and 3.4.2 are branches of 3.4) may be sufficient for the internal workings of a VCS, but giving them *names* is much more useful to humans. Figure 1.6 suggests a way we can do this.

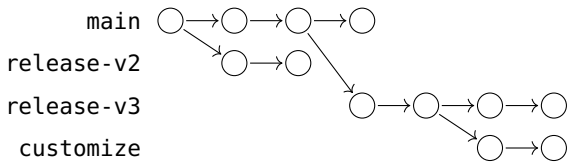

Figure 1.6: Revisions on named branches.

Instead of a trunk as in RCS, we simply start with a `main` branch[17] and create new named branches as neeed. We add commits to each branch as we go along. Since the branches are not numbered (and the naming system for commits is not yet specified), we now rely on the arrows between specific commits. This is why we have been drawing the arrows all along.

[17] Git normally calls this `master`, and Mercurial calls it `default`.

In this case, once we identified the defective, wallaby-producing commit our customer was using, we made a new `customize` branch starting from that commit, and did our two iterations to fix `kanga.c` and `roo.c`. The `customize` branch is now independent of the other branches, so it's safe to make any desired customer-specific fixes even if they break other uses of the Marsupial Maker.

What is a branch? Do they exist without revisions?

The fundamental or philosophical idea behind a branch is that it represents a *line of development*. We may make a branch for some technical or procedural reason that is not really "line of development"

oriented, but the ability to branch so as to enable separate lines of development is at the heart of any good, modern VCS. This tells us *why* we have branches, but not *what they are*.

Figure 1.6 deliberately obscures a related question: do branches exist if there are no revisions on them? This question might seem silly at first, but it's not. In fact, it's tied deeply into this same question of what, precisely, a branch *is*. Defining this generally is difficult, because every VCS has its own unique branching features and details that vary. However, Git, Mercurial, and even Subversion do all agree that creating a branch causes a logical copy of everything being branched. That is, you identify something—a file, a tree-of-files (perhaps the entire work-tree), a commit or branch name, or even the entire repository—as to-be-branched, and the system makes available two copies of that object or set of objects: one on the original line of development, and another on the new line of development. New check-ins or commits on the previous branch do not affect files on the new branch, and vice versa.

In both Git and Mercurial, you use one command to create a branch, then a second command to make the first new commit on that branch. In this case it seems as though the answer is yes: branches exist before their commits. In both VCSes, though, the actual answer is no: branches are *not* independent of commits. Branches do not exist until there are commits on them.[18,19] The two systems achieve this in very different ways, but the end is the same: the command that *seems* to create a branch may merely set up the *next* commit to create the branch.

In both Git and Mercurial, then, it seems that a branch is just a specific, non-empty collection of commits that share some particular "line of development" idea. This is at least somewhat accurate, but does not capture all the details. Moreover, as I just remarked earlier, we sometimes need to create branches for technical or procedural reasons (this is especially true in Git, which makes creating and destroying branches easy, useful, and sometimes even fun).

Note that in clone-based, distributed VCSes, cloning (or forking) a non-empty repository instantly creates a new branch, or perhaps many new branches. This sidesteps many issues and (at the cost of extra disk space)[20] makes it easy to see just *how* new commits, whether they are on new or existing branches, won't affect anyone else's work. It also makes it trivial to discard this kind of "branch": just delete the extra repository. There's nothing fundamentally wrong with this method, but we'll see how to use multiple branches within one repository.

[18] I believe this is not really a fundamental constraint: both systems could be modified—in different ways—to allow empty branches. This is how they currently work, though.

[19] We will see in Chapter 2 that in Git, a newly-created branch usually has many commits on it immediately, so that the entire question becomes a bit moot. We can still see the distinction, though, when we run `git checkout --orphan`.

[20] Both Git and Mercurial have methods to avoid using extra disk space when making local repository copies. The effectiveness of some of these tricks decreases over time, though, as new commits go in.

The other way around: are commits separable from branches?

Figure 1.6 also deliberately obscures another question: How do we know what branch a commit is on? For that matter, can a commit be on *no* branch, or on *multiple* branches? Can we *move* a commit from one branch to another? Again the answer is a bit difficult because different VCSes use different strategies. For now, let's just note these two definite answers: In Git, a commit is on zero or more branches simultaneously and the answer to "which ones" is tricky (further details must await later chapters). In Mercurial, on the other hand, each commit records its branch, so every commit is on *exactly one* branch, where it stays forever.[21] Hence Figure 1.6, which implies that each commit is on the branch whose name is on the left, works well enough for Mercurial, but not for Git.

[21] Or until *stripped*, anyway.

Commit identity

We also still need some way to identify each commit. The commits can be numbered sequentially, or there may be a *GUID*—a Globally Unique Identifier—for each commit. Sequential numbering is convenient: revision 747 clearly comes before 803, for instance. Providing sequential numbering is difficult with distributed VCSes, for an obvious reason: there's no central location to give out unique but sequential numbers.[22] In addition, two revisions being in direct sequence does not mean they're directly related: revision 747 may be on branch `release/v2`, with revision 748 on branch `main` and revision 749 on a third branch.

Exercise 1.3: GUIDs look like magic. We'll see later how both Git and Mercurial achieve them, but meanwhile, what method can you think of for turning a commit into a unique number, that will be the same on another system if and when they make the exact same commit?

[22] Subversion, which has a centralized repository, uses sequential numbering.

Git uses GUIDs: every commit has a name that looks something like `a2741b3` (but much longer); this name can be used at any time and always refers to that specific commit. GUIDs have the advantage that the same commit has the same GUID in *every* copy of the distributed repositories that have it. However, they're not very memorable, and it's not immediately obvious whether commits `a2741b3` and `04677bb` are related at all, much less whether one is the revision just before or after the other.

Mercurial uses a hybrid approach: a revision has both a sequential number *and* a GUID, such as `747:a2741b3`. The sequence number is *local to the repository:* when this commit is transferred to a different Mercurial repository, it gets a new sequence number.[23]

[23] The actual sequence number is the next one available, so two repositories that are synchronized regularly will have mostly-similar numbers. This has led some of my co-workers into believing they can identify commits by the local number, but it's not true: for instance, if Alice creates rev 747 in her repository and Bob creates rev 747 in his repository and then Alice picks up Bob's work, Bob's code will be rev 748 in Alice's repository. Meanwhile when Bob picks up Alice's work, that wiill be his rev 748, so when they both pick up Carol's latest commit, they will both number it 749.

Changesets and snapshots

Whether or not a VCS uses deltas internally, it must offer a way to *show* the difference between a pair of revisions. Ideally you should be

able to compare any two arbitrary revisions, but comparing adjacent commits—i.e., before-and-after versions of all the files—to get the set of instructions that modify the "before" version to produce the "after" version gives you a *changeset*. In other words, this is a delta or (if the change affects multiple files) a set of deltas that should be applied all together.

The original set of files, including all the unchanged files, is a *snapshot*. The new set of files, after applying all the deltas, is also a snapshot. Algebraically (ignoring tricky issues like renamed files), we can view this as two complete snapshots a and b, with their changeset being $b - a$. Note that the changeset may have less information than the two snapshots: in particular, all files unchanged from a to b produce nothing—an empty delta—when subtracted, so they are not in the changeset. However, if you have the previous snapshot and the changeset, you can always produce the subsequent snapshot, since $a + (b - a) = b$. Likewise, given two snapshots, you simply subtract (or *diff*) them to get a changeset. We will see interesting cases of converting snapshots to changesets when we consider merges and cherry picking.

Ideally, the underlying in-repo storage of a VCS should be irrelevant. Git's authors are particularly enthusiastic about its storage model, though, so Git should be thought of as using snapshots. Mercurial can be thought of either way, since it does a better job of hiding its underlying storage representation, but it actually uses changesets. In any case, both systems will present changesets and snapshots upon request.

Merging

Merging is a critical part of a VCS, giving it much of its power. Without the ability to merge, branching merely multiplies the amount of work needed. Often, we wish to bring two branches back together.[24] This may be temporary, so that previous work does not need to be replicated; or it may be permanent, so that future work need not be replicated either. Note that it is *a merge*—here the word *merge* is a noun, or an adjective as in *merge commit*—that brings together or *merges* (verb) the two lines of development. In both Git and Mercurial, for this to happen, you must have both of these lines in one repository.

[24] In Mercurial, only multiple "heads" are actually necessary for making merge commits. Git's branches are more loosely defined, and it is also possible to make merge commits in Git without using branch *names*. Nonetheless, these are mostly small semantic tweaks around the basic idea.

Let's go back to our Marsupial Maker, and see what happens once we've fixed the issue specific to our important customer. We will ignore the `main` and `release-v2` branches as well, reducing everything to just the `release-v3` and `customize` branches.

We're ready to send the fix to our customer, or may even have al-

ready sent it, but we have also been testing this fix to see if it applies generally to version 3. It turns out that it does, but only if we use just the *first* commit on `customize`. What we want, then, is to merge the applicable part of the customer-specific fix back into the release branch, keeping the second set of customer-specific changes private to `customize`.

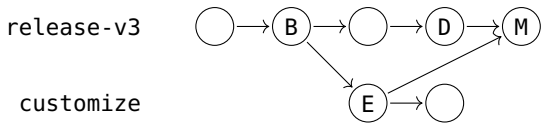

Figure 1.7: Merging.

This is shown in Figure 1.7. The merge commit is the node labeled `M`. The precise details of how this merge is done are left for later chapters. For now, note the commit labeled `B`: this is the *merge base;* and note the two commits labeled `D` and `E`. The process by which the merge commit is made is called a *three-way merge*.[25] If this three-way merge finds no conflicts, Git and Mercurial will make the new commit automatically. This works remarkably well in real-world code, despite the fact that neither program has any deep understanding of the files being merged,[26] and are instead applying simple text substitution rules.

[25] The name "three-way" refers to the fact that these three items are used to make the new fourth commit. Since only two sets of *changes* are involved—those from `B` to `D`, and those from `B` to `E`—it might have been better to call this a "two-way merge," but it wasn't.

[26] You can set up specific *merge drivers* that implement more-intelligent merges, but this is nontrivial.

Concurrency model

Whether centralized or distributed, any VCS that lets multiple users work independently of each other must offer some method for dealing with potential conflicts. As mentioned earlier, one method is *locking:* before changing a file, the user must obtain a lock, which is then released upon committing the change. This simple method has the obvious problem mentioned earlier of prohibiting parallel work. It's possible to make the locks finer-grained—Alice might lock the top half of the file, leaving the bottom half available for Bob to lock and change—but this has scaling issues. In addition, users and/or administrators must have ways to break locks since users will lock files but fail or forget to unlock them (e.g., after deciding not to commit).

If the VCS provides a *merge* model, two or more people may work on the same files, and at defined rendezvous points—in a CVCS, at check-in / commit time, for instance—they are given a chance to reconcile their changes. Merges are also needed when combining branches, and in modern DVCSes, the same methods are generally used for both of these.

Note that in a DVCS, the rendezvous point (and hence any merging) can occur *after* checkin. Bob may be able to pick up Alice's work

before doing his own checking-in, but because the system is distributed, Bob does not have to wait for Alice (nor vice versa).

Exercise 1.4: Consider this deferred merge. What advantages might it provide? What disadvantages can you think of?

What not to version

Not every file should always be versioned. Defining precisely what should be committed and what should not is tricky. Most version control systems *can* deal with non-text (data or binary) files, but not necessarily very well. Git and Mercurial in particular will both handle arbitrary files, but strongly prefer, in a sense, files that break up into lines, and files that compress well, especially against previous versions. Already-compressed files, such as many archive formats, compress poorly if at all[27]—both against themselves, and against previous versions of the same archive. Their *components*, however, tend to compress well against previous versions, especially if they are made from non-binary (text) inputs. Thus, as a general rule, it makes more sense to version-control the original input files instead of the resulting archive.[28]

[27] Technically, these files have high *Shannon entropy* measures; see Chapter 4.

The same reasoning applies to generated files such as compiled code, PDF documents, and the like. (These are sometimes called *build artifacts* today.) If you have original sources for these, along with whatever software is required to translate the sources into the finished product, it is usually best to store *only* the original sources. Of course, whatever did the translating is a key component as well, so you may wish to store the translator, or the source to the translator, or at the least, some sort of *reference* by which you can reconstruct the original translation.

[28] This will increase the number of files or other internal objects the VCS must manage. The tradeoff here still usually favors storing the originals, though.

As Yogi Berra supposedly said (though attributions suggest a Danish origin): "It's tough to make predictions, especially about the future." Similarly, it is hard to know what you will need in the future to reconstruct the past. But as a rule, the smallest possible set of sources is the most appropriate thing to keep in your VCS.

Review of some common VCSes

Table 1.3 gives a far-from-complete list of some noteworthy version control systems. It is meant only to offer a bit of flavor and insight into the history of version control, and which systems have become popular and commonplace and the features that drove it. These appear in (very) rough order of implementation.

The first two entries (SCCS and RCS) date back to time-sharing Unix systems. If there were multiple developers, they shared a single machine and there was no question of distributing a repository. CVS, a follow-on to RCS, was written to take advantage of then-new net-

Table 1.3: Some notable version control systems.

Name	Atomicity	Concurrency	Distributed?
SCCS	file	lock	no
RCS	file	lock	no
CVS	file	merge	no
ClearCase	file or commit	lock or merge	no
Subversion	commit	merge	no
bazaar	commit	merge	yes
Git	commit	merge	yes
Mercurial	commit	merge	yes

Acronyms: SCCS is the Source Code Control System, RCS is the Revision Control System, and CVS is the Concurrent Version System.

worked systems and hence the ability to share the repository—still singular and central—across multiple client machines.

ClearCase is an unusual system in that it provides multiple models and concurrency controls, and a feature called dynamic views where other users' changes show up immediately, with no explicit update-the-view action on the part of the user. The construction of the view by which file versions are selected is done with programmable rule-sets. The view rule-set file is also versioned, and it may refer to other versioned rule-sets, so that various versioning rule-sets may affect which rules are used to select a version from other versioning rule-sets, which in turn can select more versioned rule-sets, and so on.

In my view, this extreme level of programmability is something of a trap. For instance, it can become very difficult to see why you got a particular version of a file. Dynamic views also seem to be a solution in search of a problem: I prefer my working tree to remain stable until I explicitly ask the system to update it.

These systems all use file-level atomicity, which ultimately proved inferior to commit-level atomicity. Many CVS users moved to Subversion, which is very similar to CVS but features commits. ClearCase has also added commits, but after I last used it, so I have no experience with them.

Much of the Open Source world has moved on to true distributed VCSes. Git seem to be the most popular today [programmers.stackexchange.com contributors, 2014b], overtaking Subversion in 2014 or 2015. Mercurial appears to have a much smaller share of this market [programmers.stackexchange.com contributors, 2014a], but I include it here because it has generally similar abilities and features and it is instructive in its contrasts. If you can use Git, you can use Mercurial, and vice versa. Some widely used software is maintained in each system today,[29] so it is good to know both. Yet the two systems, otherwise so similar, encourage very different usage patterns. I make no further remarks on Subversion, in part because it uses a quite different model, and it is of course not distributed. I have not used Bazaar and maybe should not have it in the table (heh).

[29] Besides Git and Mercurial themselves, which are each maintained in themselves, many open-source systems are maintained in Git or are mirrored on GitHub. The original C versions of the Python language were maintained in Mercurial at the time I started this book, but have since been moved to Git.

Distributed, commit-based VCSes appear to be the path into the

future. The distributed nature of their repositories is a key feature: one simply clones an existing generally-accessible repository and begins working. Changes (changesets and/or snapshots) can be sent back to other users and other repositories in many ways, but again a key point is that the user or group who made a clone can simply publish their modifications in a new, generally-accessible repository, allowing the original authors to take or reject those modifications, and providing the modified versions to other users. We'll see later the gritty details of this process, which are slightly different in Git and Mercurial.

2
Git, Mercurial, and graph theory

Most revision control systems require that commits are stored in branches in a one-to-one fashion. If we exclude merges, these commits form a tree. Trees are well-behaved and present no real issues to branching and merging. Mercurial behaves like this, and its users might be tempted to ignore this chapter. However, the presence of merge commits convert a branch tree into a graph, which has at least one surprising (if rare) consequence. Mercurial allows you to work with the graph when necessary—admittedly not as common an occurrence as with Git.

Git, on the other hand, starts you out with more generalized graphs, with commits not neessarily bound or limited to any one branch. Git chooses not merely to expose this, but to make it a central facet of everyday use. Thus, to use Git effectively, we need to cover:

- what a *graph* is, specifically a *directed acyclic graph* or DAG;
- the *in-degree* and *out-degree* of a node in a DAG;[1]
- the notions of *predecessor*, *successor*, and *topological sorting*;
- what it means for a node to be *reachable*;
- and for merging, the concept of a *lowest common ancestor*.

[1] You won't need to remember the exact terms, but will need this concept.

You should have a good working knowledge of these by the end of this chapter.

Graphs, directed graphs, and cycles

A *graph* is simply a collection of nodes and edges that connect these nodes. Mathematicians usually use the word "vertex" rather than "node", writing this as $G = (V, E)$, meaning G—the graph—is defined by two sets[2] V (the vertices) and E (the edges). We'll mostly stick with the word "node", except when using formal math notions.

[2] In our initial graph, the edges are technically a *multiset*.

The nodes, which we'll draw as circles here, represent things that can be connected, and the edges—lines between the circles—connect them up. In our case, the edges will eventually connect commits, but let's begin with edges that represent bridges over a river.

A graph with multiple edges connecting the same nodes is called a *multigraph*. This kind of graph was first formalized by Leonhard Euler in 1736, to tackle the famous Seven Bridges of Königsberg problem. Here the nodes represent landmasses: the north and south sections of the city, divided by the Pregel River; and two islands within the river. The edges represent bridges connecting each landmass.

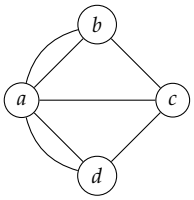

Figure 2.1: A multi-graph for Königsberg.

The original problem was to devise a walk through the city, starting on any of the land-masses, that crossed each bridge exactly once. Euler proved that there was no such walk: you must skip at least one bridge, or cross at least one bridge twice.

In Figure 2.1, node *a* is the western island. It has two bridges that lead to the northern part of the city *b*, one bridge to the eastern island *c*, and two bridges to the southern part of the city *d*. The eastern island also has one bridge linking it directly to the north and one more directly to the south. The edges—the bridges—provide a way to cross from one node (landmass) to another.

For instance, from the north (node *b*), we might cross either bridge to the western island *a*, and from that island we might cross either of the other two bridges to get to the southern main landmass *d*. We may cross any of these bridges in any direction, allowing us to reverse our path (or choose any other, of course) to go northward.

Königsburg, which was in Prussia, is now named Kalingrad and is part of Russia, and two of the bridges are gone. In particular, the western island now has only one bridge going north and one going south. (The lost bridges were destroyed during World War II, in 1941, when Lenin ordered bombing of Königsberg. Some of the remaining bridges were eventually rebuilt as well, but there are still only five bridges today.)

Removing redundant connections like this produces a *simple graph*. Unless otherwise stated, mathematicians usually mean "simple graph" when using the term "graph", and we will deal only with simple graphs below.

In a real city, these bridges are of course not actually redundant—they provide routes around traffic problems, for instance—but let's just go with it.

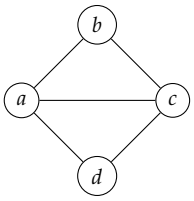

Figure 2.2: Simple graph corresponding to Figure 2.1.

A *path* through a graph is simply a walk from any one node to any other node, using the edges between nodes to make the traversal. For instance, using the graph in Figure 2.2, to get from the northern landmass *b* to the southern *d*, we have four options: cross to either island and then to our destination, or cross to either island, then to the other island, and finally to our destination. (Some definitions allow a path to loop back on itself, i.e., to walk through a node more than once. For our purposes this is not helpful so we will disallow it.) The *length* of a path is the number of edges traversed.

A *directed graph* is a graph in which all the edges are one-way links—arrows, if you will, or one-way streets in a city. If all of the Kalingrad bridges were one-way, they might form a directed graph

like that in Figure 2.3. A path in a directed graph must traverse the connections in the direction of the arrows, so in this case, to get from b to d, we could no longer go via island a, but only island c.

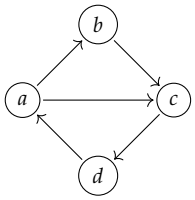

Figure 2.3: Directed graph.

If only some lanes on one of the Kalingrad bridges are closed, however, we might get a graph like the one in Figure 2.4: Now we can avoid island c when going from b to d via island a, but to get from d to b, we must pass through island c. This graph is called *mixed* and the directed edges are called *arcs*, to distinguish them from the two-way edges. For our purposes later, we will use only directed graphs and hence won't need to distinguish between edges and arcs. Some people like to maintain this distinction even with directed (unmixed) graphs, and we will do so for the rest of this chapter. (Later, though, we'll be calling our arcs "parent links".)

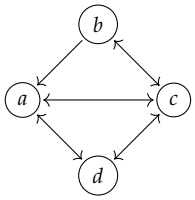

Figure 2.4: Mixed graph.

The *degree* of a graph node is simply a count of its edges. With a directed graph, we split this into *in-degree* and *out-degree*: the in-degree is the number of incoming arcs and the out-degree is the number of outgoing arcs. A node with in-degree 0 is called either a *root* or a *source*, and a node with out-degree 0 is called a *leaf* or *sink*.

Two nodes in a graph are called *adjacent* if the shortest path between them traverses a single edge or arc. In a directed graph like Figure 2.3, these adjacent nodes have a *predecessor* and *successor* relationship.[3] The defintion of predecessor and successor is very simple: One node is a predecessor of another if it is on the "before" side of the arrow, and the other node is then a successor of the first node, because it is on the "after" side. In Figure 2.3, nodes b and c are successors of a because there is an outbound arc from a to both b and c.

[3] Some prefer to call this "direct predecessor" and "direct successor", and may call d a "transitive successor" of a since we can walk from a through c to d. We do not need this fine a distinction, but keep it in mind when using directed graphs in other applications.

Need better adjectives than "before" and "after"...

A *cycle* is a path through a graph, starting at some node, that returns back to the same node without reusing an edge. The cycle is described by the nodes in the path without regard to which one comes first, so for instance, in Figure 2.2, the cycle $a \to b \to c \to a$ is the same as $b \to c \to a \to b$: these only count as one cycle.

Exercise 2.1: Does the "do not reuse an edge" constraint matter in a directed graph? Remember that a path cannot traverse the same node twice.

Exercise 2.2: How many cycles are present in Figure 2.3?

An *acyclic graph* is simply a graph with no cycles. When the graph is both directed and acyclic, we can use the predecessor/successor relationships to perform a *topological sort*, resulting in a node sequence in which all predecessors are listed before their successors. (This is impossible in a cyclic graph: for instance, in Figure 2.3, a must be listed before both b and c, and c must be listed before d, but d must be listed before a. To perform a topological sort on a cyclic directed graph there must be a way to break all the cycles. There is not necessarily any preferred way to break cycles, but in this particular case, breaking the link from d back to a produces the graph in Figure 2.5, which suffices: now a, b, c, d is one—and in this case, the only—valid topological sort for all the nodes in the graph.) Since our Git and Mercurial commit graphs are acyclic, there is always at least one

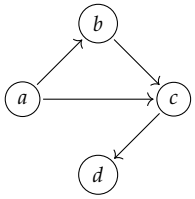

Figure 2.5: Directed, acyclic graph.

Exercise 2.3: If we remove the $b \to c$ link in Figure 2.5, how many valid topological sorts are there?

valid topological sort.

In a graph, paths also determine *connectivity*: two nodes are connected if there is a path between them. Connectivity and its corresponding paths also implies *reachability*: if one node is connected via some path to another, the second node is reachable from the first node, by walking that path. There may be multiple paths; we need only one to declare reachability.

Exercise 2.4: If node r is reachable from node s in an undirected graph, is node s reachable from node r?

The graph as a whole is called *connected* if every node is reachable from every other node. In an undirected graph, the edges connecting nodes are symmetric, so there is no concern about the strength of the connection, but for a directed graph, this connectedness property is divided into strong and weak: the directed graph is said to be *strongly connected* if every node is reachable from every other node without cheating (going backwards through an arc), or *weakly connected* if every node is reachable from every other node only once we allow going the wrong way on the one-way streets or bridges.

Any nontrivial DAG is at most weakly connected. The DAGs we use to represent commits in a repository are typically weakly connected, but disconnected graphs are allowed.[4] Our DAGs also normally have a single root commit (from which all other commits descend), but multiple roots are permitted.

[4] Generating disconnected graphs and multiple roots in Git version 1.7.2 and later is easy; it's more difficult in Mercurial.

Exercise 2.5: Do multiple roots imply disconnected graphs? What about the reverse?

Lowest Common Ancestor

The Lowest Common Ancestor problem was originally applied to (and named for use with) trees. Informally, to find the LCA of two distinct nodes v and w in a tree, we start with both nodes and work our way upward towards the root. Where these two paths join together, we have common ancestors, and the lowest—i.e., furthest from the root, closest to branch tips—of these is the LCA. This is pretty easy to visualize; see Figure 2.6. The common ancestor nodes are in grey, and the lowest is solid black.

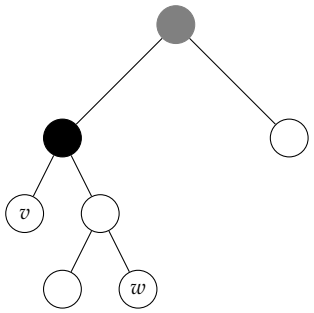

Figure 2.6: Tree LCA.

The Lowest Common Ancestor (LCA) of two nodes in a DAG is not as easy to see, nor indeed to define. Here are two equivalent formal definitions from Bender et al. [2005]:

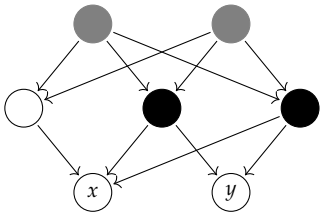

Figure 2.7: DAG LCA.

Definition 1. Let $G = (V, E)$ be a DAG, and let $x, y \in V$. Let $G_{x,y}$ be the subgraph of G induced by the set of all common ancestors of x and y. Define SLCA(x, y) to be the set of out-degree 0 nodes (leafs) in $G_{x,y}$. The lowest common ancestors of x and y are the elements of SLCA(x, y).

Definition 2. For any DAG $G = (V, E)$, we define the partially ordered set $S = (V, \preceq)$ as follows: element $i \preceq j$ if and only if $i = j$ or (i, j) is in the transitive closure G_{tr} of G. Let SLAC(x, y) be the set of the maximum elements of the common ancestor set $\{z | z \preceq x \wedge z \preceq$

$y\} \subseteq V$. The lowest common ancestors of x and y are the elements of $\mathrm{SLAC}(x, y)$.

We have not covered everything needed to properly understand either of these definitions, but there is a convenient informal (albeit slightly flawed) definition we can use: The LCA of two distinct nodes is the common ancestor of those nodes that is closest (has the shortest path) to them. That is, we measure the path length k from x or y to a candidate ancestor, and find the smallest k.

If multiple nodes have shortest paths, they are all LCAs. This is the case in Figure 2.7: both of the immediate predecessors of node y are lowest (their path lengths to x and y are 1). This cannot occur in a tree: two distinct nodes in the same tree always have a unique LCA.

Note that the LCA may be one of the nodes itself. For instance, in Figure 2.6, the LCA of w and its parent node (v's sibling) is simply the parent node. Similarly, in Figure 2.7, the LCA of x and the leftmost mid-row node is the leftmost mid-row node. If the LCA is one of the nodes, it is unique.

Exercise 2.6: One flaw is that we use a single path length metric k. Try rewriting this informal definition using two path metrics k_x and k_y. Are there more flaws?
Exercise 2.7: In a DAG $G = (V, E)$ with $|V|$ vertices (nodes), what is the maximum possible number of LCAs of any given pair of vertices? ($|V|$ denotes the vertex count.)

Aside: graphs are everywhere

Graphs and graph theory—including concepts like reachability and path lengths—are, quite unsurprisingly, used in GPS systems that provide directions. However, they also turn up in both computer and social networks. Even the neurons in your brain can be represented by a directed graph: outgoing arcs from each node (neuron) are expressed physically as synapses, which connect to the next nodes in the graph. (These use *weighted edges*, as some inputs are more significant than others, and some inputs are actually inhibitors, i.e., have negative weights. One must also weight each edge dynamically due to neurotransmitter fatigue.)

Commit DAGs

Compare Figure 2.8 with Figure 1.7. The new figure lacks the branch name labels and the letter codes, but the real key difference is that we have reversed all the arcs. That is, the newest—most recently added—nodes point back to their ancestors. This goes against the normal graph notation, so we call these parent and child relationships instead. In this case, the merge is the newest node, and we've drawn it dashed, in the midst of being added. Its outgoing arcs will point to its parents—two in this case, since it is a merge commit. These parent commits do, and in fact *must*, exist during the creation process.

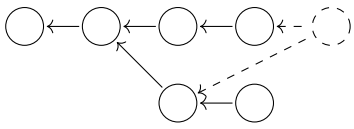

Figure 2.8: Commit DAG.

Exercise 2.8: Prove that when we start with this kind of commit DAG, and add a new commit that obeys the rules "outgoing arcs point to existing (parent) nodes and no existing node is changed", the new graph remains a DAG.

Note that we make no changes to any existing node, nor to any existing arc, when creating the new node. If we had arcs going from parents to children, adding this particular merge would require either modifying the two existing parent nodes, or keeping the lists of all arcs separate from nodes. This backwards method, with child nodes pointing to their parents, allows us to keep the arcs (parent links) together with the commit, while keeping all existing commits read-only. The fancy word for this is that they are *immutable*. (Later, we will see how this "keep parent links with each commit; existing commits are immutable" rule provides integrity checking as well as speed.)

Commit graphs, commit ordering, and reachability

We are finally ready to address a key difference between Git and Mercurial. Recall the earlier question from Chapter 1 about locating, identifying, and relating commits, and moving commits from one branch to another. In Mercurial, commits are permanently affixed to just one branch. Some of these commits may have in-degree 0, i.e., may be at the leafy ends of branches. Mercurial calls these *heads*.[5] We locate them by their branches; they define the ends of those branches. Since each commit records its parent commit identifier (or two IDs in the case of a merge), we can use these heads to reach every other commit in the branch (or indeed, in the entire graph). The DAG paths to the other commits give us their relative relationships.

Of course, Mercurial also gives us short sequential numbers for each commit, and obviously `747` comes right before `748`, making it look easy. However, even in Mercurial, two commits on the same branch, even if consecutive, may not have any parent/child relationship. For instance, commits `747` and `748` might *both* be heads on that branch. They will both be children of some previous commit, such as `746`, but the branch may fork internally. (This most often occurs when picking up someone else's work with `hg pull`.) We will see how to resolve this later; although the terminology changes, the method is the same in both Git and Mercurial.

Git uses a radically different scheme. Commit nodes *do not* retain branch information. They *do* retain their parent commit identifiers, just as Mercurial's do, but finding all leaf commits requires trawling through the entire repository.[6] To speed this up, Git provides a general form of *external reference* in a data structure separate from the graph itself. These external references include all of Git's branches (and Git's tags, and numerous other forms as well).

Git calls the commit to which a branch name points a *tip* commit. Git's branch names *do not* have to point to leaf nodes, and *more than*

[5] In a normal DAG, we would look at out-degree rather than in-degree: nodes with out-degree 0 are the leaves in our borrowed formal definition 1. Our commit DAG arcs have all been reversed, so we change our viewpoint. The word "leaf" came from the pre-reversal view, but we continue to use it here: Git calls a commit with out-degree 0 a "root commit," so calling the other ends "leaves" is reasonable enough. Git doesn't normally bother with a term for them, but for now, we need a concise way to talk about them.

Exercise 2.9: Does traversing Mercurial's commits in sequential-number order produce a topological sort? Why or why not?

[6] There are several maintenance Git commands that do this, and they take some time to run in larger repositories. Users normally never need to run these on their own, though.

one external reference may point to any given node (including leaf nodes). In effect, each external reference adds one incoming arc to its node. This provides reachability to (some) leaf nodes, but is also the reason a commit may be on more than one branch.[7] These reachable leaf nodes get us to the remaining reachable nodes, just as in Mercurial. Unreachable leaves—nodes with in-degree 0, after adding external references—may be deleted at any time.[8]

The result is that when drawing a Git DAG, we may have multiple branch names pointing to one commit, and we may have commits that (seem to) have *no* names pointing to them. We will say more about this later. For now, let's revisit Figure 1.6 with Git in mind. We move the branch names to the right, and each branch name points to the tip of that branch. To emphasize that the *position* of a commit node has little to do with which branches contain it, we may draw them anywhere convenient. The root node is contained within every branch, so there is no reason to prefer the row labeled `master`. To show how one commit can be two different branch tips simultaneously, or a branch tip commit may occur in the midst of a commit chain, we add two more Git branch-names: `A` points to the same commit as `master`, and `B` points to a commit in the middle of the `release-v2` branch. (These names are meant to be illustrative, rather than immediately useful, though `A` would be a good place to start development of a new feature that is not yet ready to be part of `master`.)

[7] It may be better to think of commits being *contained within* some branches. Git has commands with `--contains` options to see which branches and/or tags contain particular commits.

[8] Git's garbage collector, or GC, does the deletion. It obeys rules that protect items for a while, until they either get referenced or age out, so "at any time" is not quite true. You can also disable the automatic GC.

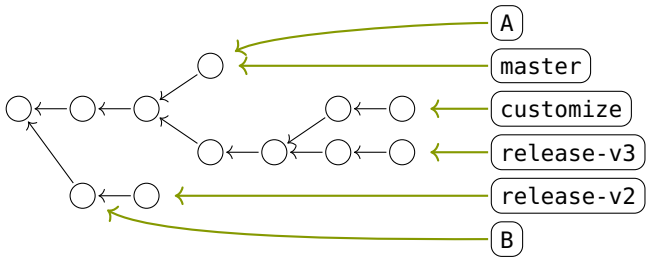

Figure 2.9: Git variant of Figure 1.6.

This brings up the other question from Chapter 1: what, precisely, is a branch? The answer is much easier in Mercurial than in Git. We can follow part of a commit DAG, starting from a (Mercurial) head on that branch, until we find the commit whose parent is on some other branch. Back in Figure 1.6, the leftmost node on `release-v3` is the first commit on that branch and the rightmost (head) commit is the last so far. The branch grows, with the head moving right, whenever we add a new commit onto that head. In Mercurial, then, each branch is its own separate entity, with the name you used when you created it, containing and consisting of an exclusive and specific (but growable) set of commits: all the commits that are made on that

branch.

In Git, the branch *name* is its own separate entity, but it is not synonymous with an exclusive and specific set of commits. The word "exclusive" is the obvious point of failure, but we must also consider the way Git is used in practice.

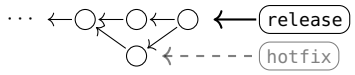

Figure 2.10: Git release with hotfix.

Suppose we create a `hotfix` branch from a `release` branch, and make one commit on `hotfix` (and prove that it fixes the bug). During that process, someone else made a new commit on `release`, so we merge `hotfix` back into the `release` branch, as in Figure 2.10. What does "the hotfix branch" mean now? Does it have just one commit, as it would in Mercurial, or does it—as `git branch --contains` contends—extend all the way back to the root? Is that hotfix commit part of the `release` branch?

Now that `hotfix` is merged back in, Git also allows and even encourages us to delete the *name* `hotfix` entirely (hence its rather tentative status in the figure). The external label `release` makes the merge commit reachable, and the merge commit makes the hotfix commit reachable, so the external label `hotfix` is superfluous. In Git's terms, the `release` branch contains the hotfix commit, so now we may as well say that this commit is on the `release` branch (and no longer on the now-deleted `hotfix` branch).

In Mercurial, we cannot delete the `hotfix` branch. It remains clear, now and forever, that `hotfix` was created from `release` and has just the one commit on it. In Git, we can—and in practice, often do—delete the `hotfix` branch *name*, but the underlying data structure—the little branch-and-rejoin sequence in the DAG—remains clear, now and forever.[9] I contend that when using Git, the word "branch" has a dual meaning: users sometimes use "branch" to mean the branch *name*, and sometimes to mean some—often vaguely-specified—portion of the commit graph. That is, Git's users know they only want *some* of the commits that Git says are contained in the branch, but are not sure how or why the commands they give Git actually select the ones they want. The branch *name* points to the *last* commit of this vaguely-specified branch, but in the most general case, it is impossible to identify the desired *first* commit. Git users often wish to find such a first commit, but Git insists that this is unnecessary. We will see in a moment how Git uses reachability and set operations to *make* it unnecessary.

[9] Or until *rebased*; but this is a separate can of worms, to be opened later.

Some users argue that this proves Mercurial to be superior to Git, because we can always trace individual commits to specific branches. Some users argue that this proves the opposite, for the same reason, noting that a statement like "commit `1417ae2` was made on `hotfix`" has no (or even negative) value several years later. I somewhat regretfully agree with the latter group, but find that this makes Git usage

more difficult and error-prone at first, because users have vaguely-defined notions of branches, vague (if any) notions about commit DAGs, and don't want to have to express subsets all the time (see the next section). Mercurial's branches are initially just right, but over time, the branch names become very cluttered. Mercurial's branch-closing feature, which hides the name from normal use, does the trick initially, but the hidden branch name still exists: you must either invent a new (often rather awkward) name or re-open the old branch, and this is where the old branch suddenly has negative value.

Understanding the way Git uses branch names within the DAG is crucial to understanding Git, so let's repeat it:

- Git calls the commit to which a branch name points a *tip*.
- Git's branch names *do not* have to point to leaf nodes.
- *More than one* external reference may point to any given node (including leaf nodes).

Refer again to Figure 2.9.

Subsetting the commit DAG

Given these commit graphs, we can and will extract interesting commit-node subsets using reachability and/or branch names. The one most users want the most often is "all commits on a branch". That is, the user says "show me the commits on `bug123`". In Mercurial, this does exactly what users want—or rather, it does until a branch name is accidentally re-used.[10] With Git, however, we return to the problem that users initially state their desires too vaguely, thinking that Git works like Mercurial. Let's see how we switch from vague to precise.

Since our commit DAG uses parent links, *reachability* implies *ancestry*: if node p is reachable from node c, p must be a parent, grandparent, or great^n-grandparent, of c. We can therefore automatically select the set of *all* ancestors of any given commit simply by selecting that commit with ancestry enabled. In Figure 2.11, selecting node y (for *yes*) with ancestry results in choosing all the nodes colored in green. Many Git commands do this by default, including both `git log` and Git's main internal workhorse, `git rev-list`. As a rule, Git commands that make the most sense with a single revision select their nodes without including ancestors, while Git commands that make the most sense with ancestry select with ancestry, and have a `--no-walk` flag to suppress the ancestor inclusion.

In Git, we will regularly use *set subtraction* on the nodes in these sub-graphs. We also may use set union. (The resulting sub-graphs

[10] This mostly occurs when different developers with different repository clones invent the same names for different purposes. Mercurial won't let you accidentally re-open a closed branch, but it's very easy for both Alice and Bob to name a branch `fix` or `for-carol`. Since branch names are both permanent and global (see Chapter 4), collaboration requires discipline with branch names. There is an extension named "convert" that can help in cases of branch-name re-use.

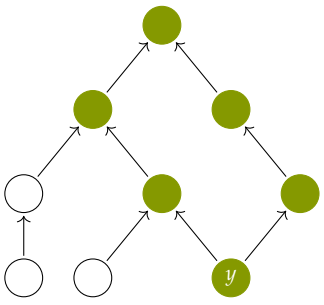

Figure 2.11: Ancestry selection.

remain DAGs and can therefore be sorted topologically as well, although they may become disconnected.) For instance, in Figure 2.12, we again ask for all ancestors of node *y*, minus all ancestors of node *n* (for *no*—and as with *y*, the ancestor set includes *n* itself). The result is again the nodes shaded in green, but this time the set subtraction operation has turned two previously-green nodes red.

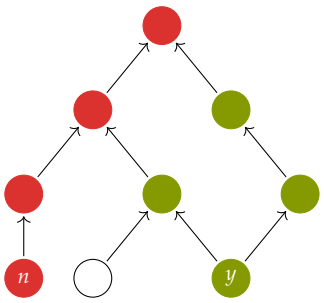

Figure 2.12: Set subtraction.

This may not seem terribly useful at first, but this kind of ancestry set subtraction is *so* common in Git that Git has a special syntax for it: `A..B` (where `A` and `B` are any valid commit identifiers). This is regularly used to request "commits I made on my local branch `B` since commit `A`", and since it resembles some programming languages' interval formats (e.g., `if i in [1..5]` to test for an integer between 1 and 5 inclusive), it looks very sequential, misleading users into thinking of it as Mercurial-style ancestry selection. Once we properly understand it as set subtraction, however, it makes sense that this excludes commit `A` itself. Figure 2.13 illustrates how `master..feature` works in Git. By selecting all commits contained in `feature` but then excluding all commits contained in `master`, we get precisely what we wanted: commits that are *only* on the `feature` branch.

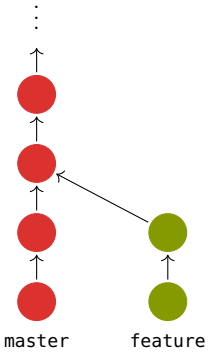

Figure 2.13: Delimiting a branch via subset.

(Did we just use the word "branch" in the Mercurial sense? No: we said we want commits that are *only* on that branch, and not on `master`. Most red-and-removed commits—at least three, assuming the vertical dots represent at least one—are on *both* branches. It's true that the first two red commits are only on `master`, so they did not have to be removed, but marking them red for a moment is harmless. We used the name `master` because the point at which the two branches join is precisely the point at which we want to stop following the ancestry of `feature`.)

One issue here is that you need to know which commit (or branch name) to select for ancestry subtraction. In particular, how did we know that `master` was the one to use on the left of the two dots? Git has a slightly hacky answer to this using the notion of an *upstream*. We will see more on this in Chapter Something.

Since Mercurial limits commits to be on just a single branch, it does not need these set operations as often. They are still needed if you use bookmarks to implement Git-style DAG-based branches within a Mercurial branch,[11] or, e.g., when you are using revision ranges to specify a large group of commits (as in `-r1200:1499`) and want to further restrict them to those within a particular branch. Mercurial has a different—richer but more complex—syntax for its set operations.

For instance, Git's `master..feature` can be expressed in Mercurial as `(ancestors("feature") - ancestors("master"))`. The two `ancestors` perform Git-style revision walks, and we subtract the

[11] If, instead of using branches, you track all your commits with bookmarks pointing to multiple heads in `default`, Mercurial ends up working much like Git.

second set from the first.

Nonetheless, Mercurial offers the *same* `A..B` syntax as Git, with a *different meaning*: select nodes that are descendants of `A` (including `A` itself) and ancestors of `B` (including `B` itself). For simple (non-branching, non-merging) chains of commits, the Git and Mercurial syntaxes select the same commits *except* that Mercurial includes commit `A`. Mercurial has an alternative spelling, `A::B`, and I find that when switching between the two systems, sticking with this second syntax reduces errors.

Symmetric differences and merge bases

Mercurial users typically (and correctly) point out that all this messing-about with set subtraction, and indeed the ability do graph-theoretic operations in general, is unnecessary in everyday Mercurial use. Moreover, Mercurial can do the same operations Git does, and many more. For instance, Mercurial's `branchpoint` operation—which selects commits with multiple children—and Mercurial's generalized descendent selection is not built in to Git (though the `git rev-list` command's `--merges` selects commits with multiple *parents*, and `--ancestry-path A^@..B` or `--ancestry-path --boundary A..B` accomplish the same thing as `A::B` in Mercurial).

There is, however, one case where Mercurial's syntax is slightly weaker than Git's: Git offers the special syntax `A...B`[12] to produce a *symmetric difference*, which is defined as the set-union minus the set-intersection of the ancestors of `A` and `B`.[13] We may think of this as commits on (or contained-in) either `A` or `B`, but not both, as shown in Figure 2.14.

This symmetric difference is particularly useful when we wish to see which commits in one fork mirror similar commits in the other fork. Git's `git rev-list` command has a number of options for examining or further subsetting commits in the symmetric difference. These are used to see, for instance, whether particular fixes have been *cherry-picked* back into a release. (We'll see what cherry-picking—Mercurial calls it *grafting*—is later.)

Note that when we perform a symmetric difference, the first commit in any excluded chain is a Lowest Common Ancestor. The lowest red node in Figure 2.14 is thus the LCA. This LCA is also called the *merge base*, as it's the point in the ancestry where the two branches join.[14] Findind the merge base (or bases) is a critical step in doing merges. Some verson control systems require users to find merge bases manually for every merge. Keeping a commit DAG allows both Git and Mercurial to find them automatically.

Recall that in a simple tree like this case, there is just one LCA.

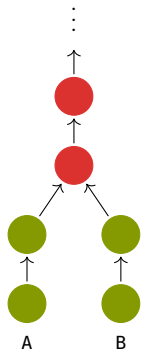

Figure 2.14: Symmetric difference.

Exercise 2.10: The `A^@` syntax is slightly tricky, and `--boundary` is also tricky. Since these details are Git-specific, we will leave them to Chapter XX. Meanwhile, what are some reasons not to just use the hash ID of `A`'s *parent* as the left operand for the set subtraction `A..B`?

[12] Note that this syntax has *three* dots, vs the usual two.

[13] Mercurial users can therefore *construct* the symmetric difference using Mercurial's built-in functions: `(ancestors("A") or ancestors("B")) - (ancestors("A") and ancestors("B"))`.

[14] Note that we have just used the word "branch" in a rather vague manner. In this case, as in most, we will assume that it's obvious what we mean—and if not, you should ask!

More complex DAGs like those in Figure 2.7 may have several. If there are multiple LCA nodes, *all* of them are merge bases. These are the "surprising consequence" mentioned at the top of this chapter. We can pick one arbitrarily, but this may lead to errors when we do the merges. Git handles this case slightly better than Mercurial, as we will see later. Fortunately, these are somewhat rare in real commit graphs.

Commit graph vs commit content

This chapter has been all about the commit graph. For any VCS to be any use at all, however, each commit must correspond to a particular source version. The VCS *must* give you some way of viewing or extracting each commit, as well as comparing commits. This is true regardless of the VCS's underlying storage model (snapshots vs changesets). This means you can—and should—think of each node in the commit graph as giving you access to a complete snapshot.

In other words, the graph itself is about the relationships between commits, while each node *within* the graph stores the metadata for a particular commit *and* a committed source revision.

This is true even for merge commits: a merge commit has a final source tree result, which may not be the same as the sum of changesets. As a particularly extreme example, suppose we were to merge two branches but tell the VCS not to make the commit yet. Then we randomly replace some files, and commit the result. The merge, in this case, has some files whose changes—as compared to either parent version or the merge-base—have nothing to do with the changes made in the branch.[15] This is not meant to suggest that you should do this—in fact, it's usually a bad idea, as we will see later. But it's always *possible*, and there are situations that might call for it.

Since a merge has (at least) two parents, each merge also gives you access to (at least) two changesets: you can compare the merge to either parent to obtain a changeset.[16] This fact matters a great deal later, especially in Git, if and when you attempt to cherry-pick a merge commit. In Chapter 8, we will go into much greater detail about the merge process: merging as a verb, *to merge*. This chapter's merges are adjectives or nouns: *a merge commit* or *a merge*. For now, let's leave this part here, and move on to Chapter 3 to look at the contents of commits, whether or not they are merge commits.

[15] This is sometimes called an *evil merge*.

[16] Since Mercurial stores changesets, you might wonder how it can store two changesets for a merge. The answer is that it stores just one changeset, comparing the merge commit's contents against the previous commit in the merge commit's branch. To get a changeset against the other, merged-in commit, Mercurial must produce it dynamically upon request, much as Git does for every commit-to-changeset operation.

3
Commits, files, diffs, and merges

As we just saw in Chapter 2, the *commit graph* is what determines the history stored in a repository. This is not necessarily the history of a specific file *within* the repository. Instead, it's the history of *every commit ever*. We should now take a look at the theory and practice of using the *contents* of each commit. That is, what is it that is *in* a commit, and how do we compare one commit to another? How do we merge branches whose contents have diverged? By the end of this chapter, you will have a better idea of what a commit does for you; you will know what a diff is, how to read one, and how to compare one commit to another; and you will have a good high level strategic view of what merges are about. If you care to dive into the details, you may learn how diffs work internally as well.

What's in a commit

Git and Mercurial have somewhat different metadata (and very different underlying storage mechanisms), but both agree to a large extent as to what a commit *is:* it is a unique entity, with an identifier (or several) to locate it. Each commit has one or more parent commits, and some metadata, including the author of the commit, when the commit was made, and a log message. Most importantly, though, these commits allow you to access—*check out*—any version of any file you ever committed.

It's worth considering here that some commits will have files that others do not. For instance, once you add and commit a new file that never existed before, no ancestor of the current commit—no earlier commit—will have the file. If you remove a file and commit, the new commit and its descendants will not have that file. So we must note that checking out some specific commit gets you the files saved *in that commit*, in the form they had at that time. If you are moving from one commit that has some file to another commit that lacks it, the VCS must *remove* that file. This means you can, in effect, go back in

time in your project, and if you go far enough back—to your very first commit—you will have only the files that were in that commit. However, all the files are in the *repository*, and when you return to the present, all the modern files return, in their modern form.

We will see much more about the VCS-assigned Globally Unique Identifier hashes in Chapter 4; for the moment, let's continue to treat them as magic. I will write them as seven letters here. This works in both Git and Mercurial, so except for these hashes being made up, these examples are realistic.

Git and Mercurial automatically get us the latest commit for some particular branch when we use a branch name, so we might run `git checkout master` or `hg co default` to get the latest commit from the main branch. Let's assume for the moment that this is commit `bcdef01`, so that this check-out step makes the work-tree match the stored content for `bcdef01`. Our work-tree now has the appropriate version of each file from that commit, and we are ready to work on it. If we need the contents of an earlier commit `a234567`, we check that one out, perhaps by ID: `git checkout a234567` or `hg co a234567`.[1] This method—using a raw hash—works in both Git and Mercurial for many purposes, and we'll use it now for illustration, without worrying about how we found these hashes, or other Git-specific checkout details.

[1] The exact spelling of a checkout argument has some side effects in Git that do not apply in Mercurial. We'll see more about this, and other ways to find or name commits, later.

Files have names

Git and Mercurial both find files by *path names* (or *pathnames*), such as `README` or `dir/file.txt`. To the VCS, a path name is essentially an arbitrary string, with slashes separating *directory* components from the final *base* file name. The path name is simply the full name, including all leading directory components. Except for the slashes and a terminating ASCII NUL (`0x00`) byte, the VCS literally imposes *no* restrictions on these path names. However, the operating system (OS) that you use may impose its own restrictions.

You may see the term *base name* used elsewhere to include the result of also stripping an extension such as `.txt` from `path/file.txt`.

The directory components of file names form a tree structure, just as we saw in Chapter 1. In general, in both Git and Mercurial, referring to a directory automatically means *all the files and sub-directories within the directory*. This process is recursive: If there is both a `dir/file.txt` and `dir/second.txt`, and a `dir/sub` sub-directory with additional files, simply writing `dir` directs the VCS to use every such file (including files in `dir/sub/deeper/` if that exists).

One common OS, Windows, uses backslash instead of forward slash in path names. Git and Mercurial essentially "prefer" forward slash internally but will convert (in either direction) for you, so you can use whichever you prefer.

Both Windows and OS X® have case-retaining, but case-folding, file systems by default. That is, if you create a file named `ReadMe`, and then ask them to open or create a file named `README`, they re-use the existing file. This same rule applies to directory names. This means you literally cannot have two files or directories whose names differ only in case. Linux and UNIX® systems, however, allow them.[2]

[2] These are generally controlled on a per-file-system basis. For instance, you can set up case-folding file systems on Linux, and fully-case-sensitive file systems on OS X. The methods, however, are beyond the scope of this book.

Git and Mercurial only store files

Note that we make a clear distinction here between *directories* and *files*. While both Git and Mercurial must provide for the existence of directories, they do not, in a sense, store the directories themselves. In particular, *neither VCS will store an empty directory*. This means that you should arrange not to require this. As a simple workaround, consider creating an empty file named `.gitignore` or `.hgignore` in an otherwise-empty directory. These files have a particular use we will see later, so your system should be able to ignore these "ignore" files and have them not interfere with real work. The presence of these files in a commit in an otherwise-empty directory will cause the VCS to create the directory if necessary whenever you check out the corresponding commit.

Character encodings

You will probably encounter, or yourself use, non-ASCII characters in path names, such as a path named `agréable/schön`. We must now take a short tour of *character encoding*. The base ASCII set consists of byte-codes `0x00` through `0x7F`. These correspond to the lower half of the ISO-8859-1 or Windows CP-1252 character sets.[3] Characters whose codes are between `0x20` and `0x7E` are all displayable and represent single letter symbols, but even simple accented characters are pushed up into non-ASCII codes `0x80` through `0xFF`. Using character sets such as Cyrillic, Greek, and especially Chinese often require abandoning ASCII entirely.

The modern answer to this problem is *Unicode*, which attempts (with some success) to provide a unique code for every symbol (and a number of emoticons or *emoji* as well) used in every human language. Unicode originally defined just over 60,000 symbols. This fit well into a two-byte encoding, called UCS-2.[4] Unfortunately, UCS-2 soon proved inadequate.

The Unicode standards currently define well over 100,000 symbols. Unicode calls these *code points*. Modern Unicode has room for exactly 1,114,112 (or `0x110000`) such code points, which it divides into 17 of what it calls *planes*, of 65536 (or `0x10000`) code points each.

[3] These are all extensions of ASCII. There are many more ISO-8859 sets as well, all numbered, and any one of them tends to be sufficient for most European langauges, but only one at a time. For instance, Czech uses letters such as z with a caron or háček, ž, that are in ISO-8859-2 but not ISO-8859-1. If you choose ISO-8859-2 so that you can spell žití (living), you lose the ability to write French quotation marks or guillemets, as in *The Tenth Doctor likes to yell «Allons-y!»*

[4] UCS stands for Universal Coded Character Set. I don't know why this is not UCCS, but I speculate that it is due to TLA (Three Letter Acronym) Syndrome.

In any case, this certainly requires more than two bytes to identify each symbol: in fact, it would logically call for 21 bits, or $2\frac{5}{8}$ bytes. Fractional bytes cause problems for many computer programs, so this would have to be rounded to three full bytes. For other reasons that seemed good at the time, though, the Unicode people actually went straight to four bytes.

Unicode's subsequent encoding, called UCS-4, still exists. It originally allowed 2^{31} code points. (It might seem like this should be 2^{32}. Originally, though, half the space was reserved—and now much more than half is reserved.) Of course, storing text this way would make every file (or file name) quadruple in length, compared to their original eight-bit ASCII or ISO-8859 encodings. Moreover, Unicode is arranged so that the most commonly used characters fit in the original UCS-2 space.[5] The Unicode committees therefore standardized numerous additional methods of encoding text.

One of these encodings, UTF-16,[6] still uses two bytes per symbol but allows *surrogates* by which some of the extra values are wedged into the available space, using two surrogate codes to stand for one code point. In effect, UTF-16 is really a variable-length encoding: using any of the surrogate codes indicates that this UTF-16 value is the upper or lower half of a pair and must be immediately preceded or followed by a pairing half. The surrogate values themselves are divided into upper and lower ranges to ensure that it's possible to know which direction to move.

UCS-4 is now synonymous with another encoding called UTF-32. It has one obvious advantage: no pair-decoding is ever needed. In practice, these days UCS-4/UTF-32 is primarily used to store strings temporarily in memory, so that indexing is simplified. To this end, when Unicode is stored in UTF-32 / UCS-4 format, any surrogate codes should be replaced with the corresponding single code point.

A third form, UTF-8, also uses a variable length encoding: code-point values in the range `0x00` through `0x7f` are encoded as a single byte version of themselves. This has the advantage that all pure-ASCII data is valid UTF-8 data. Code-points whose value is between `0x0080` and `0x07FF` encode into two UTF-8 bytes, and those whose value is between `0x0800` and `0xFFFF` encode into three. The more-rarely-used code points in `0x010000` through `0x10FFFF` take four bytes.[7] While the variable-length encoding presents some minor issues, it works very well in practice.

One problem with UTF-16 and UTF-32 is that files and other data streams are typically presented as sequences of 8-bit bytes. To send or receive a 16 or 32 bit value this way, you must pick *which* eight bits go first. This is called *endianness*, with *big-endian* meaning that you send or receive the most significant byte first. For instance, to

[5] The expanded space, using code points `0x010000` through `0x10FFFF`, is used for some less-common Chinese, Japanese, and Korean characters. It also holds historical scripts, such as Cuneiform, and the emoji.

[6] UTF stands for Unicode (or UCS) Transformation Format.

[7] The encoding allows room for up to six bytes, to represent all the original UCS-4 code points ranging from `0x00000000` through `0x7FFFFFFF`. In addition, UTF-8 can encode all the UTF-16 surrogates in three bytes, but they are nominally forbidden as well: the data stream should use the four-byte encoding for the code point for which the two UTF-16 values would be surrogates.

encode the value `0x1234` you send `0x12` followed by `0x34`. *Little-endian* is the reverse: you send this value as `0x34` followed by `0x12`.[8] UTF-16 and UTF-32 can both be either endian, and without a guide, it is sometimes difficult or impossible to tell which endianness is being used. Unicode therefore allows for a *byte order mark* or BOM at the front of a data stream. This is simply the code-point `0xFEFF`.[9] If the first two bytes of a data file are `0xFE` followed by `0xFF`, the file probably holds UTF-16-BE data: UTF-16 encoded in big-endian format. If the next two bytes after `0xFE 0xFF` are both zero, the file is probably encoded in UTF-32-BE. If the first two bytes are `0xFF` followed by `0xFE`, the file probably holds UTF-16-LE data: UTF-16 encoded in little-endian format. If the first four bytes are `0x00 0x00 0xFF 0xFE`, the file probably holds UTF-32-LE data.

The encoding for UTF-8 is strictly ordered. It is also always possible to tell from any individual UTF-8 stream byte whether it is the first byte or a continuation. As we already saw, all ASCII-compatible code points in `0x00` through `0x7F` encode to a single UTF-8 byte. All other values encode as bytes in the `0x80` through `0xFF` range, with `0xC0` through `0xDF` being the first byte of a two-byte sequence and `0xE0` through `0xEF` being the first byte of a three-byte sequence, for instance. (All continuation bytes are in the range `0x80` through `0xBF`.) In UTF-8, the BOM is therefore unnecessary, but if present, it encodes as `0xEF 0xBB 0xBF`.

In the text below, we encode Unicode characters in the standard recommended format: `U+0041` is the code point `0x0041`, which is an uppercase `A`, for instance.

[8] The names "big-endian" and "little-endian" are a nod to *Gulliver's Travels* by Jonathan Swift.

[9] The code point `0xFEFF` represents a zero-width non-breaking space, although when it is at the front of the data stream, it should normally be consumed after determining endianness. The byte-swapped code point `0xFFFE` deliberately goes unused.

Pathname encodings

Both Git and Mercurial would like to believe that all pathnames are encoded in UTF-8. This works fairly well in practice. For instance, in UTF-8, *only* a literal slash matches the directory separator slash (ASCII code `0x2F`, Unicode `U+002F`). Windows uses UTF-16-LE encoding internally, but this is invisible in normal use. Other systems mostly really *do* use UTF-8, since it mostly just works.

UTF-8 is, however, not perfect. For instance, ö may be represented as either the three byte sequence `0x6F`, `0xCC`, `0x88`, or as the two-byte pair `0xC3`, `0xB6`. The first of these represents an ordinary LATIN SMALL LETTER O (Unicode `U+006F`)[10] followed by a COMBINING DIAERESIS (Unicode `U+0308`), and the other a single letter, LATIN SMALL LETTER O WITH DIAERESIS (Unicode `U+00F6`). As far as most programs are concerned, these two file names are different, as they are made up of different byte-sequences, but they appear identical when displayed. OS X relies on *Unicode Normalization Forms* (of which

[10] The Unicode tables write these in all-capital-letters shouty form, just like this.

there are four) to normalize (i.e., convert) everything to a single common format. That way, whatever might be displayed for whichever byte-sequence you may get from elsewhere or enter at the keyboard, whether you ask to open or create `s, c, h, umlaut-o, n` or `s, c, h, o, combining-umlaut, n`, you get the same file. Linux and other UNIX systems by default *do not* normalize these, so you can have one file with each name.[11]

[11] I don't know what Windows does with these.

If you only make repositories on one operating system and only use pathnames it finds acceptable, you are unlikely to run into issues here. On Windows, you will never have both `README` and `ReadMe`. On OS X, your pretty (schön) or pleasant (agréable) file will always encode into the system's preferred UTF-8 sequence. When you *distribute* a repository, however, the fact that your directory and file names are frozen into commits[12] means that *extracting* those commits under a different operating system may cause problems.

[12] Technically, internally, they're in other, non-commit data structures in both VCSes, but the effect is the same.

Both Git and Mercurial have some kludges to attempt to work around these issues. Their effectiveness depends on what, precisely, is in the commits you would like to extract. If possible, you should avoid this situation entirely. All these problems can be corrected on Linux or UNIX systems, since the file systems are normally case sensitive and the OS itself performs no Unicode normalization. Better tools for this would, however, be useful.

Viewing file changes by comparing one commit to another

Given any single ordinary commit, our VCS should be able to tell us what happened in that commit. Moreover, given any *pair* of commits, our VCS should be able to show us the differences between them. The case of viewing a single ordinary commit then reduces to comparing that commit's parent—its immediate ancestor—to that commit. The verb for comparing two such items is *diff*, which comes from the word "difference" and refers to the UNIX diff utility.[13]

[13] The Wikipedia page for `Diff_utility` describes the command as being written by Doug McIlroy, and first appearing in 5th Edition UNIX in 1974. See Hunt and McIlroy [1975] for a description of the algorithm used in this original `diff`.

The original `diff` produced simple commands that would change one file into another: delete one particular line here, insert a different line there, replace a third line with a different third line. In other words, the output of `diff` was a set of delta-compression instructions (recall the brief discussion of delta compression from Chapter 1). *Context diffs* were introduced in 2.8BSD, in 1981; these added surrounding (context) lines, to make the diff easier to read and so that someone manually applying a diff could tell what was supposed to be in the nearby lines, i.e., whether the diff was still applicable.

In Chapter 1, we mentioned, however briefly, the notion of *file identity*. File identity is how we decide whether the `kanga.c` in commit `a234567` is the *same* file as `kanga.c` in commit `bcdef01`. It seems obvi-

ous that two files with the same pathname *must* be the same file, and usually they are—but we already noted that files also get renamed. If our VCS is to track *file history*, it must have some way to decide whether `fur.h` in commit `a234567` is related to `fur-and-scales.h` in `bcdef01`. Git and Mercurial use different schemes for this, but for now, we may simply assume that both automatically and correctly identify the files in the two commits.

If we know the two commit's IDs, we can tell the VCS: "Please diff `a234567` vs `bcdef01`." This Git or Mercurial *diff*, or difference, is simply a commit-wide comparison of all files.[14] In other words, it shows *everything* that is required to turn the first commit into the second commit. If the first commit is the parent, and the second is the child, this "everything" is *what the commit's author changed before making the commit*.

[14] As we will see later, you can produce subset diffs, but the norm is a full diff for the pair of commits.

Both Git and Mercurial show us a slight variation of what is called the *unified context diff* format, which I think is best illustrated by example. Note, however, that what we see here is not necessarily *how the author made the change*; what we see is the VCS's attempt to *summarize the result*. We'll take a look at what I mean here in a moment.

What's in a diff

At one point while writing this book, I noticed I had spelled the word "grey" in "grey kangaroo" using the American rather than Australian spelling. Since the kangaroo is Australian, I decided the Australian spelling was more appropriate. I changed it and make a new commit. The diff for the resulting commit looks like this:

The actual example here is from Git, but both VCSes display similar compatible output, and Mercurial can—and in my opinion, should—be configured to produce more Git-like output.

```
diff --git a/plates.tex b/plates.tex
index 09939ca..3dfc610 100644
--- a/plates.tex
+++ b/plates.tex
@@ -15,7 +15,7 @@ that I took on a trip to parts of Australia in February of 2010
 \end{plate*}
 The kangaroo is probably the most widely known marsupial.
 There are actually four species of large kangaroo:
-the red, the eastern and western gray, and the antilopine.
+the red, the eastern and western grey, and the antilopine.
 There are also smaller tree-kangaroos and rat-kangaroos.

 \begin{plate*}[h]
```

This diff shows what I changed in the form of a set of instructions: For now, just ignore the `index` line, and note that the `a/` and `b/` parts merely denote the before and after—or more accurately, left and right side—versions. We'll get to the `@@` line in a moment; this particular

Exercise 3.1: These instructions are line-oriented. How would you represent a word- or character-oriented diff?

one is not very interesting. Meanwhile, in the source file `plates.tex`, we may expect to find the three lines of leading context starting with the `\end{plate*}` line. Then, the original line appears, containing the word "gray". In the new version, that line has been removed, and a new line added in which the word is spelled "grey" instead. Below that, we may expect to find three more lines of trailing context. This is a total of seven (7) lines, which actually start at line 15, in both the before and after versions of this one file. This is why both sets of numbers between the `@@` symbols read `15,7`.

Let's look at one more example diff, where I simply added several lines to one file:

```
diff --git a/book.tex b/book.tex
index 1152990..5a06fec 100644
--- a/book.tex
+++ b/book.tex
@@ -43,7 +43,10 @@

 \newcommand*{\filename}[1]{\textup{\texttt{#1}}}
 \newcommand*{\branchname}{\texttt}
+\newcommand*{\remotename}{\texttt}
+\newcommand*{\rmtbranch}{\texttt}
 \newcommand*{\hash}{\texttt}
+\newcommand*{\gitref}{\texttt}
 \newcommand*{\command}{\texttt}
 \newcommand*{\setting}{\texttt}
```

The lines beginning with `@@` act as a header, marking each *diff hunk*, telling us where the changes go and how many lines to expect in the old and new revisions. If the VCS believes it knows the something else relevant that might help us read the diff hunk, it comes after the second `@@`; in this case, Git found nothing useful to include and left it blank.

In this diff hunk, I added three lines, but not all in one location. The header shows the line range: seven lines starting at line 43 in the original text, ten lines starting at line 43 in the new version. After that, the unified diff format gives three lines of leading context, two added lines, one unchanged line, one added line, and three lines of trailing context.

With an old style, non-unified context diff, we would get two diff hunks here. The unified diff format unites these (hence the name "unified") into a single diff hunk whenever the leading or trailing context can be combined, as in this case. If the added or removed lines were sufficiently far apart, however, we would see multiple diff hunks here too.

A collection of file diffs makes a changeset

Each file-level diff is a *delta*.[15] Remember also the distinction between changesets and snapshots from the same chapter, and that a changeset is a set of file deltas.

The `diff` commands thus turn commits—or rather, a specific *pair* of commits—into changesets, with one diff for every changed file. Files newly added or removed are compared against a specially-named empty file, so that all lines are either new or deleted (the diff also notes that the file is new or removed, and with the Git format, includes additional file-mode information when needed).

A changeset, which in text form is also called a *patch*, not only allows you to see what you or anyone else did in the past, it also allows you to send such changes to someone else, even if they do not have a proper VCS, or are using some other VCS. These changesets can be emailed for mass distribution, and some code-review systems use or generate emailed patches.

Git, which stores snapshots, must produce this changeset on demand. In fact, though, Mercurial produces the textual version of the changeset on demand as well, in part because its internal delta format is not line oriented—it supports binary files, which do not break up properly into lines—and in part in because it occasionally stores a fresh copy rather than a delta anyway. Perhaps most significantly, as we already mentioned, you may compare any commit to *any* other commit, not just its parent.

[15] While this is *a* delta, it is not necessarily *the* delta that the VCS stores, if the VCS stores deltas in the first place. See the remark below about binary files, for instance.

You can diff any commit against any other commit

So far, we have only compared commits to their immediate predecessors. More precisely, starting from the successor commit, we found the predecessor, then compared the predecessor to the successor. What happens if we reverse the order of the two commits? For that matter, *how* did I obtain the diffs I just showed?

The commit with the kangaroo spelling change is `c4071d9...` and its parent commit is is `a4ca39f...`. Armed with these IDs, one way to get this diff is:

```
git diff a4ca39f c4071d9
```

or

```
hg diff -r a4ca39f -r c4071d9
```

(In Mercurial, these would have different hashes, but they would also have shorter, repository-specific sequential revision numbers and we could use `hg diff -r 31 -r 32`, for instance.)

If we use this long form, it is easy to reverse the diff:[16]

[16] Git and Mercurial also offer `-R` and `--reverse` options respectively.

```
git diff c4071d9 a4ca39f
...       which produces, in part, these lines:
-the red, the eastern and western grey, and the antilopine.
+the red, the eastern and western gray, and the antilopine.
```

Note that this *reversed diff* undoes the original change. We will use this property later, to *revert* (Git) or *backout* (Mercurial) a commit.

Using the same long form, we can pick *any pair of commits* and diff the two. The output will be a set of instructions—a changeset—for turning the first commit into the second commit. This is true regardless of *how many* commits it took to get from the left-hand-side version to the right, or the time-order of the commits. If we wish to move forward in time, we must be sure to put the predecessor commit on the left, so that it is the `a/` version, with the successor as the `b/` version.

The diff is not the way the author changed things

The diff algorithms built into the version control system try to produce *some minimal changeset*. This can be surprising when we modify files with repeated text.

For instance, suppose we write a plain text file that reads:

```
the rain
in spain
in spain
falls mainly
on the plain
```

That is, we have the line "in spain" repeated twice. This is obviously a mistake, so we delete the *first* one. But then we run `diff`, and our VCS says:

```
@@ -1,5 +1,4 @@
 the rain
 in spain
-in spain
 falls mainly
 on the plain
```

That is, it claims we deleted the *second* repeated line. This hardly matters: whichever line we delete, the result is the same. But it does suggest that perhaps, things might go wrong in more interesting cases—and they do.

Suppose we start with this Python function:

```
def f(arg):
    result = []

    for i in arg:
```

```
            work(result, i)

        return result
```

and then decide we need a `prepare()` function to examine each item first, so we add that:

```
def f(arg):
    result = []

    for i in arg:
        prepare(result, i)

    for i in arg:
        work(result, i)

    return result
```

If we commit each of these and compare them, we see this diff:[17]

```
@@ -2,6 +2,9 @@ def f(arg):
     result = []

     for i in arg:
+        prepare(result, i)
+
+    for i in arg:
         work(result, i)

     return result
```

[17] Note that the diff hunk header announces that the change is inside `def f(arg)`. Both Git and Mercurial have built in rules that locate Python classes and functions, to help us view this diff.

Again, when we treat this as *instructions to change the first version into the second one*, they *work*; they are just not what we actually *did*.

A high level view of merging

The goal of a merge is easy to understand. Several people or groups, or even just one person with two or more tasks, started from a common code base, and made a series of changes. For instance, in the Marsupial Maker, Alice may be working on wombats while Bob works on kangaroos. Each person or group (or even just one person taking on multiple roles) works in her or his private repository and/or private branch. These two lines of development—i.e., branches in the philosophical sense we noted in Chapter 1—are related by this common starting point. We won't worry yet how they manage to *share* their commits, but at some point, someone—perhaps Alice or Bob, or perhaps a third person—will combine the changes. The combination should take all the good parts of both changes. The simplest method of combining is to perform the *three-way merge* from the same chapter. Now that we understand commit graphs, and have a general idea about comparing newer commits against older ones, it

Exercise 3.2: The merge result is *not* just Alice's version, nor Bob's. (Maybe we should call this Balice's—or, making note of Git's "blob" objects, maybe we could call the combination Blob's. Or maybe not.) What good would branching and merging be, if merging made the resulting source tree match one of the two sides of the merge, throwing out the other side?

is time to take a brief high level look at how both Git and Mercurial perform merges.

Two commits and a merge base

In Chapter 2, we noted briefly (see page 38 and page 45) that the LCA of any two commits is their *merge base*. In some cases, there can be more than one merge base, but this is rare and we won't address it yet. Instead, let's just note that the—presumably single—merge base is, by definition, not just *a* common ancestor of two other commits. It is, in fact, *the correct* common starting point: it is the *first* commit that is reachable from both of the two heads (Mercurial) or branch tips (Git) that we are merging. The VCS needs to find the merge base to find out both *what we did* and *what they did*.

In both Git and Mercurial, we choose one of our two commits by the normal `checkout` process. Whatever commit we have checked out now—our *current* commit—participates in the merge. We choose a second commit using some appropriate commit-identifier, typically a branch name but occasionally a hash ID, or in Mercurial, a simple revision number (or sometimes even nothing at all, and the VCS figures it out for us). That will be the "other" or "theirs" commit.[18] We may then simply run `git merge otherbranch` or `hg merge otherbranch`.

Because the VCS has the commit graph, it finds the merge base automatically, using those LCA algorithms we covered in Chapter 2. There are ways to see what commit—or, for the multiple LCA case in Git, commits—the VCS will choose, but generally we do not have to bother.

[18] I call the three commits *base*, *current*, and *other* here. Git has no single, consistent name for the current and other commits. Mercurial consistently calls them the *local* and *other* commits. I also refer to the two non-base commits as the *sides* of the merge. In several places, Git does call the current commit *ours* and the other commit *theirs*. There is, however, a problem with the ours/theirs nomenclature that we will see later, when we cover cherry-picking and rebasing.

Merge runs two diffs

Having found the merge base, the VCS then computes *two* changesets. The first one compares—i.e., diffs—the merge base against our current commit. Whatever changes show up here, those must be changes that we made, then put into commits that moved our branch forward. Likewise, to figure out what they did, the VCS diffs the merge base against the other commit. Whatever changes show up here, those must be changes that they made, moving their branch forward.

It's important for the VCS to get these two changesets right. Pretty often, it does, entirely automatically. If not, Mercurial has no way here to tweak the automated work (though it does have many manual merge *tools* that we will describe in a later chapter (XXX xref?)). In Git, though, there are a number of adjustment options. The var-

ious diff algorithms described at the end of this chapter are also available at merge time. There is one minor flaw: Git does not record the algorithm you choose, nor any options. If you ever find yourself wanting to repeat the merge, you may need to remember these. We will address some of the specifics you might want to tune in Git in a later chapter. XXX xref?

Combining changesets

The point of getting the two separate changesets is to allow the VCS to combine them. Our goal—or at least, what the VCS assumes is our goal—is to keep *one copy* of each change introduced into each file.

For instance, suppose Bob is running a merge to bring in Alice's changes. Suppose further that Alice fixed a bug in `wombat.c`, but that both Alice and Bob noticed recently that some other file (such as `doc.txt`) contained the misspelling "woombat". Both removed the extra "o", so `doc.txt` is modified (with respect to the merge base copy) in both Alice's and Bob's branches.

Both Git and Mercurial generally operate line-by-line when using these comparisons. They therefore show this change as:

```
 the ability of
-the woombat to move at high speed,
+the wombat to move at high speed,
 so that
```

(though both VCSes show several additional lines of context, as we saw earlier). Since both Alice and Bob made the *same* change to the *same area* of the *same file*, both VCSes will keep a single copy of this change.

Alice's fixes to `wombat.c`, on the other hand, have no counterpart in Bob's changes since the common merge base commit. Both VCSes can use the context of the base-to-Alice diff to find where Alice's changes should go into `wombat.c` (assuming Bob has made other changes that have moved the lines around).[19]

If both Alice and Bob modified the *same lines* within a single file, though, the VCS will declare a *merge conflict*. In this case, it will leave partial merge results in your work-tree, and you will have to finish the merge work manually (and/or with the assistance of any merge tools you like). There are several other kinds of merge conflict as well; we will address both them, and conflict resolution in general, later.

If Git believes that the merge went well, it automatically commits the result.[20] Mercurial, however, insists that you run `hg commit` to commit the merge. This may reflect the fact that Mercurial originally did not have any way to correct the merge if it was not right when

[19] In fact, neither VCS has to use the context directly. They can just count the number of lines that Bob added or deleted before the region where Alice made her changes. But "find the context, and change it there" works pretty well as a mental model of how the merge works.

[20] You can suppress this with `--no-commit`; then Git behaves just like Mercurial.

first committed (it does now), while Git has always had that ability. In any case, it's wise to inspect and/or test the result of a merge: the VCS has no deep knowledge or insight and simply thinks that the merge is good if there are no conflicts. By making you commit the merge yourself, Mercurial gives you a chance to correct it first. By committing it, Git requires that you fix it up afterward. There are pros and cons to all approaches here; we'll examine them in more detail later.

Git offers two additional actions that it calls merges, although neither one produces *a* merge. For the moment, we have not even seen how to make ordinary commits, so we will only mention them in passing:

- A *fast-forward merge* is not a merge at all. Instead, it merely moves one of Git's branch pointers (and updates your work-tree and some other Git-specific items).
- A *squash merge* performs the *merge action:* the two diffs and the combining of the results. That is, it accomplishes a merge-as-a-verb. However, it then suppresses the final commit so that you must run `git commit` manually, and once you do, the new commit is *not a merge commit*. We will address this in more detail later (XXX when?), when we get into Git-specific merges.

Note: Readers not interested in details about diff algoritms may skip the rest of this chapter.

Minimal edit distances

Git offers the option to select among several minimal-edit-distance algorithms. The algorithms in Git are `myers` (also called `default`), `patience`, `minimal`, and `histogram`. Describing these fully is beyond the scope of this book, but we'll touch on them briefly, starting with a fairly in-depth look at the minimal edit distance problem.

There is a standard (albeit impractical) dynamic-programming minimal edit distance algorithm for transforming an input string A consisting of m symbols into an output string B consisting of n symbols. These symbols may be individual characters, or entire lines. I believe the logic is clearer with characters so let's use those in this illustration. Consider for our first edit changing "bat" (or maybe "bag" or "bog") to "cart"; for our second, let's change "gull" to "gum".

For the first edit, we need to have four symbols to spell "cart". We can get there by deleting some or all of the three symbols in "bat" (or "bag" or "bog"), replacing some of the three with symbols from "cart", and/or inserting new symbols from "cart". If we assume, for

the moment, that we've correctly handled all but one symbol—that is, that we've done whatever deletion and insertion is required to get everything except, say, the first "c" correct—then all we need to do is replace "b" with "c". The cost of this is 1.[21]

When changing "gull" to "gum", though, the result will be different: we "replace" the "g" with "g". The replacement cost is free (zero). Hence, we want to define a replacement-cost function for a single symbol pair x, y:

$$\text{rc}(x, y) = \begin{cases} 0, & x = y \\ 1 \text{ (or more, up to } \infty), & x \neq y \end{cases}$$

To get to this point, though, we may have needed to do some insertions and/or deletions. In fact, since the input string in our bat-to-cart case is shorter, we will definitely have to insert something. There are of course other input and output strings that are the same length, but let's consider the cost of an insert. To insert a symbol like "c", we will use a cost of 1 as well. The appropriate cost for a delete is less obvious, but in effect, Git and Mercurial use 1 here too.

Let's step back and consider the overall problem now. We are going to take an m-symbol-long input string A, such as "bat" or "gull", and produce an output B that is n symbols long, such as "cart" or "gum". We may as well define the cost of transforming any empty string to any other empty string as zero: $\text{cost}(\epsilon, \epsilon) = 0$. Similarly, the cost of transforming a non-empty string to an empty string is the length of the non-empty string (because we have to delete all those symbols), and the cost of transforming an empty string to a non-empty string is the length of the non-empty string (because we have to insert all those symbols).

Thus, we define our base cases as:

$$\text{cost}(A, B) = \begin{cases} 0, & A = B = \epsilon & \text{empty—do nothing} \\ \text{length}(A), & B = \epsilon & \text{delete } A \\ \text{length}(B), & A = \epsilon & \text{insert } B \\ \ldots & \text{else} & \text{see below} \end{cases}$$

We also define $\text{tail}(S)$ as the rest of the symbols in string S, after stripping off the first one, S_0.

Then the *minimum* cost for solving the whole problem—along with the minimum sequence of insert, delete, and replace operations—is obtained by evaluating this recursive formula, using the empty-string ϵ cases to terminate the recursion:

$$\text{cost}(A, B) = \min \begin{cases} 1 + \text{cost}(\text{tail}(A), B), & \text{delete } A_0 \\ 1 + \text{cost}(A, \text{tail}(B)), & \text{insert } B_0 \\ \text{rc}(A_0, B_0) + \text{cost}(\text{tail}(A), \text{tail}(B)), & \text{replace } A_0 \text{ with } B_0 \end{cases}$$

[21] This assumes we have a one-unit-cost *replace symbol* directive. However, if we are inserting or replacing entire lines, we could charge a higher cost for longer lines. If we must delete-and-insert to cause a replacement, the cost seems like it should be 2.

In practice this does not really matter too much. However, in a window-display-update algorithm I modified in 1981, I computed fairly exact line-insert, line-delete, and line-replacement costs—with replacement taking any re-usable existing line contents into account—as these corresponded to the number of control codes (for insert or delete operations) and/or ASCII characters (for any visible replacement text) one sent to a display device over a potentially *very* slow serial port. In that era, 300 and 1200 baud (30 and 120 bytes per second) data transmission rates were common. The $O(n^2)$ time complexity became a problem on larger screens, though, such as the 60-line Ann Arbor Ambassador.

That is, assuming we haven't reached one of the empty-string base cases, we find the best way to do *everything else* that *would* be required if we delete the first symbol in A, insert the first symbol in B, or replace the first symbol from A to B. To this best case cost, we add either 1 (for insert or delete), or our replacement-cost-function based on the first-symbol equality. This adds nothing if we're getting a free replacement because the symbols match.

For Git and Mercurial diffs, we discard the notion of replacing a symbol (or a line): we only delete and insert. In effect, the replacement cost if the symbols do not match becomes infinite, so that the minimum cost comes from inserting or deleting instead. We keep the symbol—the character or line—if and only if it matches.

As we noted at the beginning of this section, one way to solve this without re-evaluating all these intermediate results (which would give us an explosion of recursion) is to use the *dynamic programming* technique, where we make a data structure to hold problems solved thus far and avoid re-solving them.[22] We create an $m + 1 \times n + 1$ matrix M representing two items: a *best edit cost*, and the chosen action-thus-far that *produces* this best-cost. The value in $M_{i,j}$ represents the best cost of transforming the first i symbols of A into the first j symbols of B. We fill the upper edge of the matrix (row zero, $i = 0$) with the cost of converting an empty source string to as much of the destination string as that cell represents. We fill the left edge (column zero, $j = 0$) with the cost of converting that much of the source string to an empty destination string:

$$M_{0,j} = j \quad 0 \leq j \leq n$$
$$M_{i,0} = i \quad 0 \leq i \leq m$$

Except for $M_{0,0}$, which represents *stop*, each top-edge, row-zero entry corresponds to an insertion of the $j - 1$th symbols of string B; similarly, except for $M_{0,0}$, each left-edge, column-zero entry corresponds to a deletion of the $i - 1$th symbol of string A. We can remember these as arrows: a left arrow $\leftarrow$ means *insert* and an up arrow $\uparrow$ means *delete*. (We'll see in just a moment why I use these particular arrows.) Hence the initial 4×5 matrix for changing *any* three-symbol string such as "bat" to *any* four-symbol string such as "cart" is:

$\times 0$	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$
$\uparrow 1$	$-$	$-$	$-$	$-$
$\uparrow 2$	$-$	$-$	$-$	$-$
$\uparrow 3$	$-$	$-$	$-$	$-$

and, for instance, $M_{0,4}$, which is 4, is the cost of inserting "cart" if we were to initially have the empty string (which of course we don't), while $M_{1,0}$, which is 1, is the cost of deleting "b" from "bat" or "bag"

[22] This is functionally equivalent to *memoization*: for any two string parameters A and B, remember the cost to transform string A to B, along with the chosen operation—insert, delete, or keep-or-replace—in a cache. However, the matrix method is particularly elegant.

or "bog" so as to leave the last two letters (which is of course not our ultimate goal, but may serve to get us towards it).

Next, we simply fill all the remaining rows and columns using the minimum of the three operation costs.[23] For matrix element $M_{i,j}$, the minimum cost is:

- an insert of B_{j-1}, whose cost is $1 + M_{i,j-1}$ (one step left), or
- a delete of A_{i-1}, whose cost is $1 + M_{i-1,j}$ (one step up), or
- a replacement of A_{i-1} with B_{j-1}, whose cost is from the replacement-cost function plus the cost of getting here through $M_{i-1,j-1}$ (one step up and left).

[23] In case of ties, it doesn't matter much which we pick here, though for screen updating, "replace" is less jarring visually and should win. Note that whenever the symbols at A_{i-1} and B_{j-1} match, though, the do-nothing "replace" choice always wins.

The curious thing is simply recording *the source of the cost* (up-and-left, up only, or left only) suffices to find our minimum edit *path*, once the matrix is completely filled. For instance, let's watch the matrix get filled with directions for changing "bat" to "cart", using our arrows to each matrix entry to show where the cell's calculated cost comes from.

The initial table becomes, after a pass through the first row:

$\times 0$	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$
$\uparrow 1$	$\nwarrow 1$	$\nwarrow 2$	$\nwarrow 3$	$\nwarrow 4$
$\uparrow 2$	$-$	$-$	$-$	$-$
$\uparrow 3$	$-$	$-$	$-$	$-$

(Each entry in row 1, except for $M_{1,0}$, is a *replace:* replace "b" with "c" at $M_{1,1}$, for instance. This is because there is no "b" in "cart".)

The last two rows finish making it interesting:

$\times 0$	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$
$\uparrow 1$	$\nwarrow 1$	$\nwarrow 2$	$\nwarrow 3$	$\nwarrow 4$
$\uparrow 2$	$\nwarrow 2$	$\nwarrow 1$	$\leftarrow 2$	$\leftarrow 3$
$\uparrow 3$	$\nwarrow 3$	$\uparrow 2$	$\nwarrow 2$	$\nwarrow 2$

Note that this matrix forms a Directed Acyclic Graph.

Here, for instance, $M_{i=2,j=3}$ represents an insertion. The symbol to be inserted is B_{j-1} or "r". However, we start at the lower right corner of the matrix, $M_{3,4}$. This points up and left, i.e., represents a replace operation, that replaces $A_{i-1=2}$ with $B_{j-1=3}$. Since both are "t", we do nothing at all, leaving "bat" alone. The total cost of this is 2, although we have not paid it yet: this cost of 2 is inherited from above.

We then move northwest in the matrix, to $M_{i=2,j=3}$. This points left-only, i.e., represents an insertion. The symbol to be inserted is B_{j-1} or "r", so that we now have "bart". We follow the arrow left to $M_{i=2,j=2}$ where we find the cost is now reduced to 1 and the arrow points up-and-left: "replace". This replaces $A_{i-1=1}$ with $B_{j-1=1}$, so that "a" stays "a" and we continue to have "bart". Now we move to

$M_{i=1,j=1}$. This has another northwest arrow, so we replace $A_{i-1=0}$ with $B_{j-1=0}$, changing "b" to "c", producing "cart".[24] We follow the arrow up and left, arriving at position $0, 0$ and terminate.

[24] When we modify the algorithm to forbid symbol replacement operations, we will insert the "c" and delete the "b", moving left once and then up. Note that this simplifies interpreting the matrix, since diagonal arrows now always mean "keep".

Longest Common Subsequence and the Myers algorithm

Converting a simple string like "bat" to "cart" with our original algorithm uses a four by five matrix.[25] The compute time is therefore $O(mn)$ in the number of symbols. For two versions of a file with about ten thousand (10^4) lines each, a diff would have to make about 10^8 comparisons, and this is far too slow to be practical.

[25] One can shrink this to 3×4 since the zero-edges are so easy to calculate, but this makes little difference.

However, we can see intuitively from the matrix that long sequences of *symbols match exactly; move diagonally* have no added edit cost and usually result in a winning path. This is one reason to forbid symbol replacement in favor of insert and delete only: now diagonal paths always mean "symbols match and can therefore participate in LCS". Then all the diagonal transitions represent common (matched-up) symbols. If we compute the entire matrix and find the best path through it, the symbols retained through diagonal movement make up the *longest common subsequence* or LCS. Unfortunately, finding *the* LCS is itself computationally hard.

There are several algorithms that do better than $O(mn)$, yet always find the LCS. For instance, one is known as the Method of Four Russians.[26] This divides the large notional matrix into smaller t-blocks and uses offset vectors and the observation that the difference between any adjacent matrix cells is at most 1, allowing us to avoid allocating and computing some parts of the larger matrix entirely.

[26] The "Four Russians" name is due to the four authors of a paper on construction of transitive closure of a directed graph. While the authors' names—Vladimir Arlazarov, E. A. Dinic, Aleksandr Kronrod, and I. A. Faradzev—seem likely Russian, according to Wikipedia, "It is unclear whether all the four authors were in fact Russian at the moment of publishing the paper."

Even this is still impractical for diffing large files, but there are numerous heuristics that do work well in practice. For instance, assume that many—not just a few—symbols really do match up. We may be able to use this to identify some single, reasonably long diagonal in the matrix we might build from A and B. Imagine, for instance, that there are 9000 untouched lines in the 10000 lines in our hypothetical file. These lines—the symbols in A and B—need not occur exactly once in each file, but finding them, then alignining the remaining non-unique-but-also-matching lines, is *much* easier when there are unique lines: We start at the unique matches, then extend outwards in either diagonal direction.

Now we can simply divide the input-and-output strings (or files) into two parts. We will only build and look at the upper left sub-block (or sub-box) of the full matrix that comes before our long diagonal sequence, and the lower right sub-box that comes after. We then recursively compute two sub-diffs on the parts before and after the long sequence. If our hypothetical 9000 lines are right in the middle,

we immediately reduce the problem from roughly 10^8 comparisons (the entire $10^4 \times 10^4$ matrix) to an upper-left 500×500 matrix and a lower-right 500×500 matrix, and now we need only $2 \cdot 500^2$ comparisons. Furthermore, if these sub-matrices have a long diagonal sequence within them, we will win there again. Perhaps in the end we will only compute three or four 10×10 matrices for three or four changed regions. This is a classic divide-and-conquer strategy.

Git uses an algorithm due to Eugene W. Myers [Myers, 1986], which typically runs in $O(ND)$ time. Here D is the length of the edit script—the number of insertions and deletions—and N is the length of A and B, which are assumed to be roughly the same length. (If they are wildly different, D is guaranteed to be large.) It assumes that there are many matching symbols (lines) and uses a greedy algorithm to find the best available long diagonal, so that we can divide-and-conquer as above. I will leave the complete details of the Myers algorithm to the cited paper, but as long as D is relatively small, which it usually is, this diff algorithm is *much* faster. Git's implementation adds some extra heuristics that accept sub-optimal diagonals early in some cases, to avoid extremely slow behavior if D is large.

Mercurial uses a customized internal diff that makes some fairly brash assumptions about long common subsequence matches, choosing long ones "near the middle" of the two strings for its divide-and-conquer approach. Since there are no user-tunable parameters anyway, there is not much more to say about it.

Git: minimal, patience, and histogram diffs

The `minimal` algorithm simply modifies Git's `myers` implementation to avoid the heuristic shortcuts, and thus compare more items when it looks for long common sequences. The goal here is to spend more compute time to make sure that the edit sequence is as short—i.e., minimal—as possible (but still not necessarily the most *sensible*, as we have seen). Since these shortcuts give Git's diff a lot of its speed when Git is comparing very dissimilar files,[27] this really does slow down the diff. It is not at all clear when this produces shorter edit scripts in practice.

[27] How common or rare this is depends on the inputs.

The `patience` algorithm uses a different technique. First, it makes a pass over the input files to find which lines in each file are really unique and which ones repeat. The idea here is that a lot of programming languages' source files have sequences of easily-matched "noise lines", e.g., consisting of just the word `end` or a closing brace or similar. Treating these as long common sequences is not helpful. If anything, it is liable to cause false matches, where the diff de-

cides that several close braces or `end`s in a row can be matched up to make a shorter edit script that replaces one or two "significant" lines instead. Hence this variant applies the greedy algorithm (for long common sequences) to whatever remains only after these non-unique "noise lines" are stripped out. It then extends the best diagonal so far with the original "noise" lines re-inserted as long as they continue to match. Finally, it divides and recurses on the sub-boxes (finding unique lines anew, since each sub-box may now have unique lines that the original did not).

The `histogram` algorithm is a further modification of the `patience` algorithm. Instead of taking *only* unique lines to find long common sequences, it counts the relative *frequency* of the lines. Less-frequent lines "count more" in the long common subsequence, but unlike `patience` diff, the lines need not be entirely unique.

This last algorithm should, in theory, probably be the best for at least some files such as source code. However, some internal hash functions were mis-applied in a way that causes some files to see pathological behavior from histogram diff. This is fixed in Git version 2.12. If histogram diffs do prove to be better in general, Git will probably make these the default algorithm eventually.

4
Distributing repositories

One of the distinguishing features of Git and Mercurial is that they are distributed (as noted in Table 1.3). We have said what this *means*: that there is no central repository, so that no clone is any more special than any other (except by user choice of course). We have not yet, however, said anything about how this is achieved. By the end of this chapter, you will understand the key principles underlying the distribution of repositories. This includes the use of both names (such as branch names) and *hashing*, although the use of the names differs in Git and Mercurial.

We noted earlier that Mercurial branch names are global. Global, in this case, means *the same in each repository-clone*. It's easy enough to see how branch names work: when Alice creates a branch named `for-carol` and Carol synchronizes with Alice to get Alice's work, Carol acquires both the branch and the commits. Git's branch names, however, are deliberately *not* global, or perhaps can be called semi-global, which is possible mainly because Git's commits are not tied to particular branches.

We also noted that both Git and Mercurial use GUIDs—Globally Unique Identifiers—to identify commits: commits have names like `a2741b3...`. Both systems use these GUIDs to discover and exchange commits whenever you direct the system to synchronize your clone with a peer. In order to make this work correctly, these GUIDs really must be globally unique.[1] It would not do for Bob to create a *different* commit (in Git) or changeset (in Mercurial) and wind up with the *same* GUID that Alice got for the changes she gave to Carol, because then Carol's system would believe that she already had Bob's work.

Both Git and Mercurial produce these GUIDs by *hashing* commits. We will see how both the hash function and the data given to it are critical to make this work. In fact, Git hashes almost everything you give it, and exposes more of this to users: given a GUID (i.e., hash value), Git makes it trivial for you to view the item that was hashed. You will sometimes see this said as "Git stores content" (which is

[1] More specifically, they must be unique among all clones of a given repository, *including forks that may rejoin in the future*. This is a somewhat weaker requirement than true global uniqueness. For instance, if Alice makes a commit, but then destroys it without ever sharing it with anyone else, the destroyed commit is allowed to have the same GUID as some future commit, or a commit in an unrelated repository. You can think of this as allowing Doppelgänger commits: they may share a GUID only as long as they never meet.

true, and is also useful for some mistake-recovery, although "storing content" has a deeper philosophical meaning we'll see later). Mercurial keeps its hashing better-hidden, so that the only place you see it is in changeset GUIDs. In this chapter, we'll look only at what both VCSes do in common, in order to make repositories distributed.

Hashing

Hashing, in our case, is the process of taking some input *key*, e.g., a string, and mapping it to a *hash value*, such as a smallish integer. That is, we define some function $h(k)$ to take large inputs and reduce them to smaller outputs. Hashing can get very complicated, with all kinds of requirements on hash functions, the ability to do multiple stages of hashing, and so on.

If the input keys have more possible values than the output hash value, the Pigeonhole Principle tells us that at least some *different* inputs must result in the same hash. That is, there are keys k_1 and k_2 with $k_1 \neq k_2$ for which $h(k_1) = h(k_2)$. These are called *hash collisions*.

Almost all hash schemes must deal with collisions. One exception is the so-called *perfect hash*, where a known, fixed set of input keys will be mapped to the output hash values. In this case the image of h may be smaller than its codomain, i.e., we might allow 45 input keys to map to 55 hash values; or h may be surjective, if we can figure out how to map our n keys into exactly n values. Perfect hashes are often used to decide whether some arbitrary input is actually one of the pre-selected keys, and if so, which one. They are particularly useful for operations like turning computer language keywords into internal token-IDs.

The paragraph above uses the terms *codomain* and *image*, which are more precisely defined than *range*. In this chapter, though, we'll use the word *range* to describe the cardinality of the set of all possible output values of our hash function, assuming arbitrary keys.

In our case, our hash function should:

- be deterministic;
- accept arbitrary length data;
- be reasonably fast;
- have a defined range;
- have good uniformity; and
- be non-invertible (also called *one-way* hashing).

The first property, determinism, is required of any hash function. That is, if h is any hash function and we have two keys k_1 and k_2, $h(k_1) = h(k_2)$ whenever $k_1 = k_2$. (Note that this says nothing about the hash function value when the keys differ.)

The second property, arbitrarily long input keys, is a requirement because both Git and Mercurial hash the contents of at least some version-controlled files and/or changesets. Both of these have sizes bounded only by the underlying operating system. Here, both systems have borrowed hashes from cryptography, because cryptographic hashes operate on very long keys. Cryptographers call their inputs *messages*, and their outputs *message digests*.

The third property, speed of computation, is to some extent a matter of compute power and taste. If we are to compute many hashes, though, the speed (or lack thereof) of the hash function will affect the usability of the VCS. Mercurial computes one hash per added

changeset plus one hash for new files.[2] Git computes a hash for virtually everything, so speed of hashing is more important in Git, but it is still significant in Mercurial.

The fourth property, a defined range, allows us to store the hashes in fixed-size fields. Mercurial makes somewhat heavy use of this property internally (but is good about not exposing it). Git also uses it internally, but currently exposes its fixed-size hashes directly by requiring that some scripts (called *hooks*) often spell out Git's *null hash*,[3] and documenting the form and length of hash values. (Mercurial's hooks are written in Python and use libraries that hide the hash's details.)

The fifth property, uniformity, has to do with avoiding collisions. Informally, a hash function that has good uniformity will produce seemingly-random output values, spread across all possibilities, from different inputs. This is perhaps most easily illustrated by considering several extremely-*non*-uniform hash functions operating on integers, such as $h(x) = 0$ (or any other constant), or when the output range is, say, integers between 0 and 1023 inclusive, $h(x) = x \bmod 2$. Even if the output does use all possible values, though, it might still be biased, such as $h(x) = \lfloor \log_2 x \rfloor$ (with input values ranging from 1 to 2^{N-1} and outputs from 0 to $N - 1$). A hash function with good uniformity uses all output values and is not biased like this.

The last property, one-way hashes, is not strictly required. We *want* it because we will use the output of the hash function as our GUIDs. A one-way hash means no one can deliberately create a file or changeset that produces the same GUID as something already in the VCS.[4] That is, even with malicious intent, no one can cause our VCS to fail. Without this property, someone could (at least potentially) engineer such a failure.

Both Git and Mercurial currently use the cryptographic hash function known as Secure Hash Algorithm 1 (SHA-1), which produces a 160-bit hash. SHA-1 is part of a group of algorithms denoted by number: SHA-1, SHA-2, and SHA-3. SHA-2 and SHA-3 both comprise multiple functions with 224 or more bits of output (see Dang [2015]). These functions are generally referred-to by their bit length, with SHA3 as a disambiguator if needed: e.g., SHA-256 vs SHA3-256. All of these hash algorithms have large yet uniform outputs and are thus plausible candidates for GUID generation (see the end of this chapter for the gory details).

Avoiding accidental hash collisions

Of course, we would like to avoid *accidental* failures, so let us consider the probability of hash collisions. The chance of a random (non-

[2] The notion of "new" here is admittedly fuzzy.

[3] This could have been avoided early on by, e.g., providing an option to `git hash-object` or `git rev-parse` to produce the special null hash, and the hash for the empty tree. The current plan for expansion is to use the fact that Git allows one to *abbreviate* hashes, so that if Git moves to a longer hash, shorter hashes might still work as before. I foresee some problems with this plan, but time will tell.

Exercise 4.1: The second case declares that the output range is more than just one bit. If the output *is* just one bit, does x mod 2 provide good uniformity? Consider whether we know anything about the input keys assigned to x.

[4] More precisely, while it's technically possible, the amount of computation required is overly expensive.

engineered) hash collision depends on three factors: the size of the output value space (the range r below), the uniformity of the hash function, and the number of keys hashed.

As we just noted, SHA-1's output is a 160-bit message digest, which both Git and Mercurial encode as 40 hexadecimal digits: these are the `a2741b3...` values you see as GUIDs. This provides a huge range of encoded hashes: 2^{160} is a bit over 10^{48}, or about 1.46 quindecillion (using short scale names for numbers). To get a better handle on that, consider that 10^{16} is ten quadrillion, and 10^{48} is ten quadrillion squared (10^{32}) times bigger. A larger hash output space would of course provide even more range.

Obviously $\log_2 2^x = x$, and $\log_{10} 10^y = y$. Since logarithm bases scale, we can first compute $\log 2 / \log 10 \approx 0.30102$ and then simply compute $160 \times 0.30102 = 48.163\ldots$ here.

Mercurial provided enough room in advance to switch to a 256-bit hash such as SHA-256, but there has not yet been a need to do so. One might think that, at least in the absence of malice, even SHA-1's 160-bit digest is nuke-it-from-orbit overkill; but it is trickier than that. We will examine this in some detail in an optional section below, but for now I will note that you may see the figure 2^{80} or 10^{24} elsewhere. This is the number of messages you would have to hash to get a 50% chance of collision. Presumably we want more than a 50/50 chance that our VCS will cease to function. To get 99.9999999999999999% reliability—that's 18 nines—we can use a much smaller number, 1.7×10^{15}, as our limit.

Note that this number—over 1 quadrillion—implies very large repositories. 10^{15} is close to 2^{50}, and since each digest represents a unique message, we need roughly 2^{50} distinct messages just to get a 1-in-10^{18} chance of an accidental collision. We don't know how big these messages are (between the actual average object or changeset size, the compressions applied, and general overhead, it is not obvious), but even if it were just one byte—and it is definitely more than that—the 2^{50} factor alone implies we need at least a petabyte just to store all the inputs. There seems to be no danger here.[5]

[5] The performance of both Git and Mercurial tends to fall off as the number (and size) of items in the repository grows, limiting repositories to numbers far smaller than these. Of course, this also depends on a reasonably uniform hash function and—in the presence of potential mischief—the one-way property.

How DAG + GUID = distributed

Before we dive into the process below, let's make a clear distinction between *private* and *public* commits. A private commit is simply one that you have not shared with anyone else. Each commit has a unique ID, so those that are private have IDs that no one else has. A public (or published) commit *is* shared. Git does not retain this property directly—though we will see in a while how Git's *remote-tracking names* get close enough—but Mercurial (as of version 2.1) does, calling these *phases*. (In fact, it splits *private* into *secret* and *draft*; we'll see what this means in more detail later.) Given that we have distributed repositories, any particular pair of repositories may have

some public commits that are shared between them, some public commits that are not—some commits that are shared, just not with each other—and some commits that are purely private. Our goal here is to share some or all commits, but we'll start with a slightly simpler task.

Imagine that you and Alice both started the day with perfectly synchronized repositories. It is now the end of the day; and to simplify even more, let's also suppose at first that you were stuck in meetings, but Alice was able to get some good coding done, and has committed the new code into her repository. You would like to pick up her work. You could simply re-clone her entire repository, of course: "delete the project and download a fresh copy," as Cueball suggests in the xkcd comic. If you're both on a fast network (or sharing a machine) that may even be a reasonable approach. But what if you're in different offices, with a slow network and a large repository? Or, what if instead of being in meetings all day, *you* got some good coding done too, and have your own commits? You could add the "save your work" part of Cueball's advice. Instead, though, let's see how you can pick up Alice's commits, making them public and shared with you.

"... And we're all going to stay *in this meeting until we figure out why no one is getting any work done!"*

Since each commit has a GUID, you can tell your VCS to get in touch with Alice's VCS, using the computer network as a sort of telephone call. (Both Git and Mercurial have multiple built-in network protocols for doing just this.) Your VCS then asks hers for commits. At this point, your VCS could hold a simple and straightforward conversation with Alice's, where for every commit, Alice's VCS says "I can give you the commit identified by `12345...`." Yours then replies with either "No thanks, I already have that" or "Oh yes! Please send that one!" This would be kind of a stripped-down clone operation: look at every commit, skip all the ones you already have, and bring over the ones you don't. This is an improvement over a full re-clone, but the price is quite a few have/want sequences: one for every commit in her repository. You and Alice both have a commit DAG, though, and we can do better—in fact, *much* better.

Again, for the moment let's assume that your repository is strictly behind Alice's: you were in sync this morning; she has added commits, you have not, and you just need those commits. All Alice's VCS needs to know, then, is: *What are the tip commits on your branches?*[6]

[6] Reminder: these *tip* commits are the ones that Git branch names point to, or the ones called "heads" in Mercurial.

Given the GUIDs of these commits, Alice's VCS can walk *her* commit graph from *her* tip commits back to these nodes. Because nodes are immutable and their IDs are universal and global, you and she must necessarily have *the same* graph from these points back to any root nodes. The nodes in between—the ones Alice's VCS visits on its walks back to the shared nodes—are precisely the commits Alice's

VCS should send to yours.

This is fine if your repository is strictly behind Alice's, but what if you have commits she doesn't? The picture here is more complicated. What we would like now is to find the Least Common Ancestors. Since each repository has different additions to this morning's graph, neither VCS can do this on its own. The actual implementation uses the basic have/want protocol we just mentioned: Alice's VCS gives ours an initial list of GUIDs, and our VCS tells her which of those we want. Her VCS then uses the wanted commits' parent links to offer more GUIDs, which our VCS replies to in the same way. Note that during this process, Alice's VCS carefully sends us her GUIDs in a topologically sorted order, working from tips backwards. As soon as our VCS replies with "I already have that one," Alice's VCS knows that we have that one *and* all its parents. Alice's VCS can therefore stop traversing that part of the DAG and move on to other not-yet-known-to-be-shared GUIDs.

Having identified the commits to transfer, her VCS now merely needs to send over a series of changesets, or anything equivalent. Git uses multiple mechanisms, but typically it saves deltas[7] into what Git calls a *packed archive* or *pack file* (and more specifically a *thin pack*). Mercurial packages changesets into what it calls a *bundle*. The exact details are not critical here—most users need not even be aware of pack files at all, though Mercurial bundles are useful later—but this is what is going on when you see Git's "counting objects" and "compressing objects" messages.

[7] Remember that deltas are the building blocks of changesets. Due to its unusual mechanisms, Git packs up the deltas themselves, rather than actual changesets.

As soon as you have Alice's commits, those commits are published. That is, your repository has stored those commits with their GUIDs. Alice cannot take them back—not without your cooperation, at least. We'll see in detail *how* to retract commits later, and more importantly, *when* and *why*. For now, just note that by retracting an *unpublished* commit, you can avoid sending out non-functioning work. Moreover, you can retract a broken commit and then add instead a correct commit, and no one but you need ever know. . . assuming, of course, you did not publish the broken commit already. Once the commit is lodged in other repositories, it will keep coming back when you sychronize with them.

This kind of retraction, especially with subsequent corrected commits added, is usually called "history rewriting". Some VCS users say you should *never* rewrite history. I stand with those who say there is nothing fundamentally wrong with history rewriting. If your work has become public, though, rewriting creates a number of issues. We'll go through these in detail later. Most of them are problems for your co-workers or colleagues, and you and they simply need to agree in advance as to what may be rewritten.

Push, pull, fetch

The abstracted VCS conversation above, where your VCS picks up new commits from Alice's VCS, only operates one way: you get her work. The term for this action is a little bit problematic, because Git and Mercurial use different verbs. In fact, they started out with the same verb, *to pull*, but the Git programmers combined this with merging. In Git, *pulling* gets the changes and then attempts to integrate them immediately, and the verb we want is *to fetch*: retrieve commits, with no additional processing. Mercurial defined pull the way we want, obtaining new commits but *not* merging them. Unfortunately, Mercurial has an extension that adds the verb *fetch*, with the meaning *pull, then merge, then commit*, which is what Git's *pull* verb means! For the moment, we will use the word *pull*, but keep in mind that in Git we will `git fetch`.

If you can pull, you should be able to *push*, and sure enough, both Git and Mercurial allow you to push changes. Pushing generally requires more permission than pulling: for instance, public repositories (as on github.com or kilnhg.com) allow anyone to pull or clone, but not anyone can push changes to them. If you do have permission, though, the process of pushing works much like the process of pulling, except that after your VCS dials up its counterpart over the Internet-phone, yours takes over the role of offering GUIDs and theirs takes over replying with "want" or "already have". The remote repository can also decide, independently of this commit DAG construction phase, whether to allow the push based on whatever rules the recipient chooses. We will see much more about this later, but for now, note that the pulling process is simpler: your VCS assumes that you mean to allow all the new commits in. Since they are not yet integrated into the work-tree,[8] it's quite safe to bring them all in: you can inspect them as much as you like, then take or ignore them, since your private repository is yours to deal with however you like.

Exercise 4.2: List some reasons you might not want anyone to be able to push to a repository you set up on one of these public-access sites.

[8] Unless, of course, you're using `git pull`. This is one reason to use `git fetch` instead. We will see, later, when you might use `git pull` for convenience.

DAGs, heads, and branch tips, oh my!

Pulling (or fetching) and pushing updates a commit *graph*, but we need more than just the graph. We need to be able to find the new branch *tips*.[9] We noted in Chapter 2 that Git and Mercurial use different methods for this: Mercurial automatically finds all *heads*, while Git uses branch-names to point directly to commits, making those *become* branch tips. Whenever you retrieve commits from another repository, or send your commits to another repository, what happens to these automatic heads or branch-name-identified tips? Most users find the action in Mercurial to be clearer and simpler, so we

[9] Of course, we already have not just the parent links, but also all other commit/changeset metadata, along with any necessary file names and contents, all of which were part of the thin-pack or bundle. We just need to see how these branch tips work.

will cover it first.

With Mercurial, pulling from a peer like Alice's repository—recall that a peer is any other remote repository—brings over all the changesets the remote shows, via the abstracted VCS conversation we just reviewed. (Remember that changesets can be marked *secret* on the remote. You can add your own restrictions as well, such as bringing over only changesets belonging to one particular branch, but for now let's work with the default action.) This may cause new branches to spring into existence, if the new changesets (commits) are on new branches. In any case, though, it may cause branches to contain an internal division, resulting in multiple heads, as shown in Figure 4.1. The top row of commits represents your work, and the second row are commits you picked up from Alice.

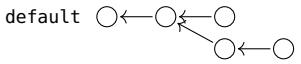

Figure 4.1: Mercurial: two heads in one branch

This Git-style branch—we might call this a branch within a branch, though Mercurial itself just sticks with the term "heads"—happens when both you and Alice made commits (Mercurial changesets) based on the point at which the fork occurred. Now that you have all three changesets, both your latest commit and Alice's two commits descend from the same common ancestor. Mercurial handles this just fine on its own: your branch now just has two heads. *You*, however, must take care of this, usually by either rebasing or merging. We'll look at this in detail later.

Suppose that either instead of, or in addition to, pulling Alice's work, you were to push your work to Alice, In this case, *her* repository would wind up with the exact same internal fork, although we might want to draw the resulting commit DAG with your single changeset on the second row,[10] and her two changesets on the first. By default, though, Mercurial simply refuses such a push, telling you that this would create a new head. You can force the push anyway, and no real harm comes of this,[11] but Mercurial is trying to encourage you to pull, then rebase-or-merge, and only then push. If you were to do this as a merge, the changesets you would push will then present just a single head (see Figure 2.10, for instance; note that the merge commit would point back to both your single commit and Alice's two).

[10] Which row a commit-sequence appears on is not significant topologically, but it is a useful visual cue.

[11] Your collaborators may get a bit annoyed, though: with one head in a branch, it's clear where to continue working and make the next commit, but with two or more, which head is headier?

In Git, though, branches are more loosely defined: sometimes we mean branch *names*, pointing to the tip of a branch, and sometimes we mean commit-DAG subsets, starting from the tip identified by a branch-name and working back to a root commit, or to some cutoff point, vaguely- or explicitly-specified. When we do a fetch or push, we acquire or send new commits, changing the commit DAG, but what about the branch *names*?

Ever since version 1.5, Git's answer for `fetch` has been to use *remote-tracking names*. Git documentation calls these *remote-tracking*

Historical note: this change, which went into full effect in Git version 1.5.3, is when `git fetch`, rather than `git pull`, became the proper counterpart to `git push`.

branch names, but I think this phrase is more confusing than *remote-tracking names:* it means we must at least sometimes refer to our own branch names as *local* branch names. For now, let's do that as necessary.[12]

In any case, to fetch, you direct your Git to contact a peer Git, such as Alice's repository, by some name that you find short, convenient, and memorable. Git calls the name you use here a *remote*. The spelling of this name is up to you, but for now let us spell this remote `alice` (you will often see `origin` instead; we'll see why later).

When you use `git fetch alice` to pick up Alice's work, you get the same commit DAG as you would with Mercurial, but since Git requires that we have names pointing to tip commits, what Git does here is to construct *new* names from the branch names Alice has, prefixed with the name of the remote itself.[13] That is, if Alice has two branch-names `master` and `test`, our Git renames these to remote-tracking "branches" named `alice/master` and `alice/test`. This is how Git achieves the "semi-global" names we mentioned at the beginning of the chapter: you see Alice's branch names, but *qualified with a prefix of your choice*.

What this means in practice is that instead of Figure 4.1, we get Figure 4.2, which is an entirely normal case of git branching (cf. Figure 2.9). Instead of just *local* branch names pointing to tip commits, we have both local names and these new *remote-tracking* names pointing to tip commits.[14]

For `git push`, though, Git does much the same thing as Mercurial, for an even stronger reason. The remote peer will by default refuse a push if it would, in Mercurial's terms, create another head. This is because Git *cannot* create another tip under a single reference name, and Git's `push` mechanism, unlike Git's `fetch`, has no built-in concept of renaming branch-names. Your Git simply asks the remote peer to set its (the remote's) branch label to point to whatever new commit you give it, forgetting where it used to point. If this new tip commit causes the peer's commit graph to lose reachability for some of its commits, those commits become eligible for true deletion. We will cover this ground again later in more detail. For now, just remember that forcing a `push` can cause your remote peer Git to discard commits.

[12] There's another reason not to call these remote-tracking *branch* names, which we will see later in Chapter 5. Specifically, checking out a remote-tracking name results in a "detached HEAD," while checking out a branch name does not. This makes the remote-tracking name significantly non-branch-like.

[13] The *full* names of these are `refs/remotes/alice/master` and `refs/remotes/alice/test`. Local branch names have a separate namespace beginning with `refs/heads/`. This guarantees that even if you have a (local) branch whose name starts with `alice/`, the remote-tracking names won't use the same full name.

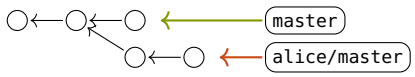

Figure 4.2: Git: local vs remote names

[14] You might wonder how remote-tracking names get updated. The simple answer is that they are updated on every fetch and push, using the branch information coming from the remote. Due to changes over time in design decisions, this simple answer is *too* simple: the precise details depend on your version of Git. Nonetheless, a good way to think of this is that remote-tracking names remember where the branches on the remote were, the last time we checked.

Automatic corruption detection and Merkle trees

Git and Mercurial both guarantee[15] that the hash of a distinct string—such as the contents of a source file—is unique. This hash acts as a *checksum*, verifying the source file's contents, as well as being a unique fingerprint identifying the contents. That makes sense for

[15] With whatever probability we achieve by limiting the number of items in the repository, anyway.

files, but we hash more than just files: we hash *commits*. The trick here is that the hash of any particular commit (in Git) or changeset (in Mercurial) not only uniquely identifies that commit or changeset, but also uniquely identifies the entire history *leading to* that commit or changeset.

Both systems begin by hashing the contents of files, specifically the files in the first commit.[16] Next, they hash the initial commit *using all the file hashes* and the work-tree layout as part of the metadata for the initial commit. This gives them the GUID for that initial commit. Since the hash depends on every input bit, and the input bits include the file checksums, file names,[17] and tree setup, the VCS can simply check whether re-hashing the root commit—whether extracted to a new work-tree, or simply as stored in the repository—matches its GUID. If so, everything is intact. (If not, the VCS cannot on its own help you reconstruct the data, but if the repository has been distributed there is probably a good copy available somewhere.)

[16] The details differ between Git and Mercurial but the overall process works out the same.

[17] Recall from Chapter 3 that some UTF-8 encoded file names may use different byte sequences on different operating systems. As you might suspect, this causes all kinds of interesting problems.

Next, for each subsequent commit or changeset, both VCSes build the new GUID by hashing not just the new file-and-tree contents (Git) or changeset (Mercurial) itself, but also the new commit's metadata, *including the GUIDs of its parent commits*. In other words, the GUID of the *second* commit depends on the GUID of the root commit. Changing the root commit changes its GUID, which changes the second commit's GUID. Similarly, assuming the third commit is in linear sequence (i.e., neither a merge nor a new branch off the root), its GUID depends on the GUID of the second commit. Changing either the root or second commit changes the second commit's GUID, which changes the third commit's GUID. The fourth commit depends on the third, and so on. The GUID of a merge depends not only on the merge's result, but also on the GUIDs of *both* parents.[18]

[18] Or all parents, for Git's octopus merges. We'll leave octopus merge for later.

This kind of sequence of dependent hashes is called a *hash chain* when it is linear, or a *hash tree* when it is hierarchical and forms a tree. (The term *hash list* is also used when there is no hierarchy involved. In our case, the hashes are in a DAG and perhaps should be called a hash DAG, but it is still called a hash tree.) Hash trees are also called *Merkle trees* after their inventor, Ralph Merkle.

Random facts I found interesting: Bitcoin uses Merkle trees to protect transaction history. Curiously, while the block chains use SHA-256, the transaction signatures use elliptic curve cryptography, which has a different approach to one-way hashing. In any case, Merkle trees are agnostic to the underlying hash.

In short, each commit GUID is not only a global identifier, but also a verifying checksum, not just of the specific commit, but of the entire history leading to that commit. This means both Git and Mercurial can and do check data integrity with every repository extraction. Of course, verifying one particular commit may not detect silent corruption elsewhere in the repository, but both VCSes have maintenance commands to examine and thereby verify every commit.

Note: Readers not interested in details regarding hash collisions, whether accidental or malicious, may skip the rest of this chapter.

Hashing and accidental collisions

We define a uniform hash function as a function $h(k)$ such that for any key k, the probability of producing any one particular $h(k)$ from the set of all possible hashes, whose output range $r = |\{h(k)\}|$, should be about the same as the probability of generating any other output hash. That is, each hash output is used with a frequency of $1/r$. This means that given two distinct keys, the chance of a collision is also $1/r$. We will use the probability of *uniqueness*, i.e., the complement of the probability of a collision, so that we may multiply probabilities as we iterate over keys. We call the probability of a collision p, so its complement, in this case $1 - (1/r)$, is $\bar{p}$.

There are a number of ways to quanitfy the overall probability of uniqueness (and hence probability of hash collisions). I uses the method below as it is rather elegant, and seems sufficient.

If n is the number of distinct keys $k_0, k_1, \ldots, k_{n-1}$, the probability that *all* keys are unique is:

$$\bar{p}(n) = \prod_{k=1}^{n-1} \left(1 - \frac{k}{r}\right) \tag{4.1}$$

Each term in the product in Equation 4.1 is the probability that the hash of the k-th key is unique, i.e., the complement of the probability of a collision with any prior hash. The first key $k = 0$ is automatically unique, and for subsequent keys, we assume there are k unique prior hashes occupying the range r, so we have a k/r chance of colliding with them. The complement, $1 - (k/r)$, is the probability that this key results in another unique hash. The overall probability is then the product of each individual probability.

(This is usually written with a constraint $n \leq r$, since if $n > r$, the Pigeonhole Principle guarantees a collision. When $n > r$, though, we get $1 - (r/r) = 1 - 1 = 0$ for the term with $k = r$, which forces $\bar{p}(n)$ to zero. The $n \leq r$ constraint is therefore unnecessary. On the other hand, the assumption that all prior hashes are unique introduces a bit of error, since any prior non-unique hashes open up more of the range. We ignore this since an earlier collision is just as much of a problem for us as a collision for key k. However, the fact that each term is, on its own, a slight *over*-estimate helps make up for the other issue noted below. In any case, after we make our approximation substitution below, we will ultimately find a much stronger constraint: we will want n to stay much smaller than r.)

Observe that as the number of keys grows, the overall probability

of uniqueness for all our hashes shrinks geometrically. In theory we could simply compute $\bar{p}(n)$ exactly, even for large values of r and n, but it is easier—and ultimately more useful—to use an approximation. We note that for $x \ll 1$, $e^x \approx 1 + x$. (The value $(1 + x)$ here is just the first two terms in the Taylor expansion of e^x at zero.) This means that in Equation 4.1, we can replace $1 - (k/r)$ with $e^{-k/r}$:

$$\bar{p}(n) \approx \prod_{k=1}^{n-1} e^{-k/r} \tag{4.2}$$

At first sight this may not help, but note that for $x \neq 0$, $x^a x^b = x^{a+b}$. The product of all of these e^a terms is just e raised to the sum of the terms. We also know $\sum_{k=1}^{n-1} k = n(n-1)/2$ (this identity is very common in big-O analysis of algorithm runtimes, for instance). Hence:

$$\begin{aligned} \bar{p}(n) &\approx \prod_{k=1}^{n-1} e^{-k/r} \\ &\approx e^{(-1/r)\sum_{k=1}^{n-1} k} \\ &\approx e^{(-1/r)n(n-1)/2} \end{aligned} \tag{4.3}$$

Simplifying the exponent in Equation 4.3 slightly gives the closed form approximation:

$$\bar{p}(n) \approx e^{-(n(n-1))/(2r)} \tag{4.4}$$

We must, however, also note here $e^{-x} > 1 - x$ when $x > 0$. When we replaced $1 - (k/r)$ with $e^{-k/r}$ we increased each term's value slightly. Since we are computing our margin of safety, raising the value of each term, however slightly, overestimates the safety of each added key. As long as we keep n small with respect to r, the error is certainly small (the remainder polynomial from the same Taylor expansion together with the Mean Value Theorem tells us that each overestimate here is $(e^\xi/2)x^2$ for some $0 \leq \xi \leq x$), but I need input from a real mathematician to say more about it. For now, the equations below use inequality rather than approximation.

Using Equation 4.4, we can produce several more-useful equations. For instance, given any particular fixed hash range r and desired chance of avoiding collisions $\bar{p}(n)$, we can find the maximum number of keys n before falling below our allowed safety margin:

$$\begin{aligned}
\bar{p}(n) &< e^{-(n(n-1))/(2r)} \\
1/\bar{p}(n) &> e^{n(n-1)/(2r)} \\
\ln(1/\bar{p}(n)) &> n(n-1)/(2r) \\
2r\ln(1/\bar{p}(n)) &> n^2 - n \\
0 &> n^2 - n - 2r\ln(1/\bar{p}(n)) \qquad (4.5)
\end{aligned}$$

Once we choose our target value for $\bar{p}(n)$, we can just write it in as a constant U ($0 < U \leq 1$). The right hand side of Equation 4.5 becomes a standard quadratic equation of the form $ax^2 + bx + c$ with $a = 1$, $b = -1$, and $c = -2r\ln(1/U)$. We need only the positive root from the usual $(-b \pm \sqrt{b^2 - 4ac})/2a$ expression, so:

$$n < \frac{1 + \sqrt{1 + 8r\ln(1/U)}}{2} \qquad (4.6)$$

For instance, if we want to find the number of keys n that gives about a 50% chance of an SHA-1 collision, we set $U = 0.5$, giving $\ln(1/0.5) = \ln 2$ (for concreteness, $\ln 2 \approx 0.693$), and set $r = 2^{160}$. Plugging these in to Equation 4.6:

$$n < \frac{1 + \sqrt{1 + 8 \cdot 2^{160}\ln 2}}{2} = 1.4234 \times 10^{24}$$

which is pretty close to 2^{80} ($2^{80} = 1.2089\ldots10^{24}$). That is, our maximum number of keys before a collision becomes at least 50% likely is about 2^{80}: just half as many bits as in the hash function.

In fact, from Equation 4.6, we can see that—as in this case—whenever $\ln(1/U)$ is small, but not vanishingly so, the range r dominates in the square root expression. Since $\sqrt{2^N} = 2^{N/2}$, whenever there are N bits in the hash output, we become more likely than not to get collisions after hashing about $2^{N/2}$ keys. In cryptography, the term *collision resistance* is defined as a variation of this property: that we cannot find *any* pair of distinct messages M and M' for which $h(M) = h(M')$ without doing about $2^{N/2}$ work. All cryptographic message digest algorithms, including SHA-1 and SHA-256, are designed to have good collision resistance.

Of course, if our hashes are so important, we would like a much-better-than-50%-chance reassurance that our GUIDs will all remain unique. Plugging in higher values for U reduces the maximum number of keys (or messages) n even further: as U approaches 1 (from below), $1/U$ approaches 1 (from above) and therefore $\ln(1/U)$ approaches zero.[19] This eats away at our range r as $\ln(1/U)$ starts to vanish. We can express our collision safety[20] in terms similar to error rates quoted—albeit not actually achieved—for storage media, such

[19] In fact, we can approximate this—and in the process see the approach to 0—for x near 1 using another Taylor expansion, this time of $\ln x$ at 1. The first two terms are $\ln 1 + (1/1)(x-1) = x - 1$. Hence for $x = 1 + \epsilon$, $\ln x \approx \epsilon$.

[20] The term *collision resistance* seems natural here as well, but it is taken. "Collision safety" is not a defined technical term, just I something made up here for convenience.

as 10^{-18} (see, e.g., Rosenthal [2010]). For our 160-bit SHA-1, to obtain this margin of safety we need to limit the number of keys to about 1.71×10^{15} or 1.71 quadrillion. Raising the safety margin by an additional factor of 10 (setting $\bar{p}(n) = 1 - 10^{-19}$) reduces our maximum number of allowed keys to about 541 trillion, which is about a factor of 3. (These much-lower values for n also reassure us that $n \ll r$, so that our approximations are good.)

Hashing and deliberate collisions

Loosely speaking, an *invertible* hash is one where, given some particular message M and hash output $H = h(M)$, it is easy to construct some message M' for which $h(M') = H$. In cryptography, this idea is formalized into two properties: *preimage resistance* means that it is difficult to find *any* message producing a known digest, while *second preimage resistance* means that it is difficult to find a *second* message producing the same known digest as an existing, known first message. If the digest is used as a signature corroborating a message, preimage resistance means the Bad Guys cannot construct a message just from the signature—this could be the actual original message, or not—while second preimage resistance means that the Bad Guys cannot replace the *real* message with a *fake* one, even if they know the real message.

In our case, we are mainly concerned with this second preimage resistance: if we have a procedure for finding M' in any reasonable time, we can disrupt the proper function of both Git and Mercurial by finding a new message that produces a hash collision with an existing Git object or Mercurial changeset. Note, though, that finding a hash collision for *any* two files is sufficient to cause problems for Git.

(The actual failure mode for Git is—or at least, is intended to be—that the new object is simply not stored, regardless of whether it is locally-generated or is brought in over the wire by push or fetch. Thus, to get a more serious failure, the Bad Guy must insert it into the repository *before* the real object goes in. In Mercurial, changesets with non-unique IDs can still be added locally, but will no longer transfer to other repositories.)

SHA-1 originally appeared to satisfy all of our conditions. However, in 2005, one group of researchers [Wang et al., 2005] showed a method for constructing collisions under SHA-1 in less time than originally expected (by about a factor of 2^{11}), and another [Kelsey and Schneier, 2005] showed a method for constructing second preimages using less work than originally expected. As of 2014, SHA-1 is no longer approved for United States Federal digital signature pur-

poses (see Dang [2012], p. 11 and Barker and Roginsky [2015], p. 14). It is now possible, albeit expensive, to produce a deliberate SHA-1 collision [Stevens et al., 2017]. The example PDF produced in 2017 does not break Git because Git adds a prefix to each blob, but the same technique could be used to produce a deliberate collision. In any case, SHA-256 still provides enough bits to be considered secure.

Note that poisoning either Git or Mercurial is not as easy as finding *any* message M', since it must have a form that the VCS will see as a valid object or changeset. Certainly, in the absence of deliberate attacks, SHA-1 suffices for both unique IDs and corruption detection. However, the fact that SHA-1 is not as secure as originally thought does suggest that Mercurial's provision to allow for SHA-256 was a good idea.

5
Basic setup and viewing

Now that we have the basic concepts of DAGs, branches, and commits, and what it means to share a repository, we would like to create, share, and clone some repositories. Unfortunately, there are several things we need to set up first.[1] We will at least get to clone the repositories for our version control systems, though, and by the end of this chapter, you will be able to do some basic configuration and viewing of a repository.

[1] There are many details and stumbling blocks here. If you are already very familiar with the shells and text-file editors available on your operating system, this chapter should be easy. If not, it may be quite frustrating.

Configuration mechanisms

Both version control systems have three separate configuration mechanisms: *configuration files*, *environment variables*, and *command-line options*. The third is the simplest, since command-line options apply to the command you just entered and therefore override everything else. For instance, with the `git log` command, the option `-p` sets the "show a patch" mode. You will only want this sometimes, so you will only specify `-p` sometimes. Similarly, `hg log --color auto` sets the "use color" option to automatically detect when to use color: but you probably want this every time. If you had to *specify* this every time, that would be inconvenient, so there are more permanent—or persistent—ways to specify configurations.

One might think a single persistent configuration mechanism would suffice, and it probably would. It would certainly be simpler to explain. But it is not, in general, how programs behave on the systems on which Git and Mercurial grew up, and they now take advantage of this—so we must delve into the topic of environment variables.

When you log in to the system, or start a Terminal session in a window, you get a command line interpreter, which the system calls a *shell*.[2] The shell prints a prompt, such as `bash$` or `sh-3.2$`, and lets you enter commands. Each command you run gets its own *environment*, which that command automatically copies out to commands

[2] You can choose which shell you prefer from those available for or on your system. Common shells include bash, csh, dash, ksh, sh, tcsh, and zsh. Most shells share features such as using dollar-sign `$` to denote *shell variables*, and asterisk `*` for *globbing*, which we will define later. Some treat other characters specially. For instance, several use exclamation points to access *shell history*. Some use the curly braces `{}` for special purposes and may therefore "eat" them where you might not expect this. You will need to learn which characters need special quoting in whichever shell you choose. As a general rule, the backslash `\` works to quote any character, so if your shell eats braces and you want to print an asterisk inside braces, you might enter the command `echo \{\*\}`.

it runs. The commands themselves form a *process tree:* when Git runs commands `A` and `B`, and `B` runs command `C`, Git sits at the top of this particular tree,[3] passing an environment to `A` and `B`, and `B` sits atop `C`, passing an environment to `C`. Another way to view this tree is as matryoshka (Russian nesting dolls). Processes higher in the process tree are "outer" and processes lower are "further in."

[3] In fact, your command-line interpreter is really at the top, in charge of Git.

Each process can *change* its own environment; from that point forward, these new settings are passed to new, further-in commands that the outer command starts. Once started, however, each command has its own private environment. No inner command can affect any outer one, nor can it change the settings in something it has already started. Hence if you set an environment variable in your shell and then run a command, that command inherits this setting.

Most shells allow you to set a variable for the duration of a single command (and hence any sub-command it runs):

```
var=value command arg1 arg2 ...
```

Again, this might seem a bit silly: if the command inspects an environment variable to affect its behavior, why not provide the command with an argument that affects its behavior instead? There are two reasons to use an environment variable, though, one of which should be clear enough: suppose the command *you* run, such as `git show`, runs some *other* command, such as `less`. If the sub-command reads its environment, you can "smuggle" some settings through the outer `git show` command into the inner `less` command and Git does not have to know that they exist, much less what they do.

The other reason to use environment variables is to affect commands in more obscure ways, where either no command option is available, or you wish to affect several commands in a row. To accomplish the latter, you set the environment variable using a shell built-in command:

```
export var=value
```

(the syntax varies a bit in some shells). Next, you run the several commands. Last, you restore the previous environment, so that no additional commands are affected:

```
unset var
```

or re-export with its previous setting.

It is pretty common in *shell scripts*, which are scripts the command line interpreter can run by spinning off a *sub-shell* process,[4] to set environment variables for the duration of the entire script. When the script is done, the sub-shell terminates. The outer shell's environment

[4] Scripts should indicate *which* shell interprets them with their first line, which resembles a comment in most shells and which all shells know how to skip. Hence most scripts begin with a line reading `#! /bin/sh`. You can write scripts in other languages as well, e.g., beginning one with `#! /usr/bin/env python` to write one in Python rather than shell. The reason for using `/usr/bin/env` here is to allow the `env` command to find the Python interpreter, whose location in the file system tends to vary.

is unaffected, since this is a separate process. This same technique works everywhere, including in Python programs. (Mercurial is written in Python and it is sometimes useful to write Python code to extend Mercurial in various ways.)

Any settings that you wish to retain permanently, across multiple logins or in separate Terminal windows, you should, of course, save to configuration files. This is what we will do now.

Configuring your identity

When you create commits, both Git and Mercurial save your identity as the *author* and/or *committer*. These are part of the metadata stored with each commit.[5] We'll see how to view this metadata in a moment. For now, let's see how to set it.

Both Git and Mercurial use the same identity format: your real name, and your email address. For no particular reason, Git splits this into two configuration entries, while Mercurial uses one. Neither system will check whether you are telling the truth about user name or email. If you want to claim to be Barack Obama, who are VCSes to say you are not?[6] Replace the user name and email address here with the one(s) you want to use:

```
git config --global user.name 'Your Name'
git config --global user.email 'email@example.com'
    or
hg config --edit
    then add
[ui]
username = Your Name <email@example.com>
```

(For Mercurial, you may want to read ahead for a bit to find out how to choose your editor while using `hg config --edit`.) From now on, unless overridden by a less-global setting, the VCS will use this as your identity. You can change these at any time: a new `config` command (or editing the configuration) will change the stored value. This will not affect any *existing* commits; only new ones will pick up your new identity.

Incidentally, this general form—the name of the VCS, either `git` or `hg`—followed by a verb, then options and/or additional arguments, is how both VCSes are set up today. Very old versions of Git used instead a `git-` prefix, so instead of `git config`, you would run `git-config`. Git's documentation still works this way: the documentation for `git config` is named "git-config", for instance.

[5] Git stores a separate author and committer, while Mercurial stores just one name. Git's method allows for separate accountability with, e.g., emailed patches. Mercurial simply assumes that all authors have direct access to the repository; if you will make a commit on behalf of someone else, it is up to you whether and how to put the author's information into the commit.

[6] Later, we'll see how to digitally sign your work. This will allow other people to test whether a commit with your name on it is really something from you, or from some imposter.

Additional configuration

For most users, I recommend setting both the pager (usually to use `less`—you may want to add options such as `-S`, as shown here), and automatic colors as well:

```
git config --global core.pager 'less -S'
git config --global color.branch auto
git config --global color.diff auto
    or
hg config --edit
    then add
[extensions]
pager = LESS="FRSX" less
color =
```

The color remarks below assume you have enabled color. The pager setting here for Mercurial includes the `-F`, `-R`, and `-X` options in the environment, whlie the setting for Git does not, because Git has special code that puts `LESS="FRX"` in the environment, provided the environment variable is not already set.[7]

There is one other value you may wish to configure immediately, which is the command that the VCS should use to open your favorite editor. In Git this is the `core.editor` setting, and in Mercurial this is the `ui.editor` setting.[8] If you do not set it, the VCS falls back to other methods to choose an editor (the specifics vary depending on your system). Once you have set it, though, you can run `git config --global --edit` or `hg config --edit` to open up this particular editor on your user-specific configuration file, which will let you edit particular entries to fix typos without having to rerun command-line commands:[9]

```
git config --global core.editor 'your chosen editor'
git config --global --edit
    or
hg --config ui.editor='your chosen editor' config --edit
    then add to the [ui] section
editor = your chosen editor
```

Try these out to make sure the editor starts correctly. This will also show you the layout of the configuration files, which should be fairly obvious. You can now fix any typos in your name and email address, for instance. Note that the short form name `section.setting` converts to the longer `[section]` and then `setting = value` form in both VCSes.

Git and Mercurial both have three levels of configuration setting.[10] Unfortunately, the two VCSes use different options to select among these, with some of the option names matching but with different meanings. (I get them mixed up when switching between VCSes.)

[7] I honestly have no idea *why* Git does it this way, but note that if you want to *prevent* this `FRX` setting, you can set the environment variable to something. Of course, command line options override, so you can, e.g., set `core.pager` to `'less -S +FRX'` to clear them—but you probably *do* want `-FRX`.

[8] Mercurial used to let you update your configurations using `hg config` without invoking an editor, but now requires that you run `hg config --edit`. This results in a chicken-and-egg problem: how do you tell Mercurial which editor to use until you use an editor to edit the configuration? The secret is to use `--config ui.editor=editor`.

[9] You only need quotes around the editor argument if you are supplying option arguments as well, or if the path to the editor contains spaces, but the quotes won't hurt in general.

[10] Git adds a fourth level, `--file`, which is only needed if you are going to write Git scripts of your own and want to borrow its configuration code but not use any standard configuration file.

- Machine-level: all users on a shared machine. To select this in Git, use `--system`; in Mercurial, use `--global`. You will only need this option if you are a system administrator, configuring settings that affect all users of a shared machine.
- User-specific: settings for you, when you are logged in. To select this in Git, use `--global`. In Mercurial, there is no option to select it: you get it by not specifying *any* option.
- Repository-specific: the current repository only. To select this in both Git and Mercurial, use `--local`. If you select no option, this is Git's default, but not Mercurial's.

The repository-specific setting is particularly useful if you want different email addresses for different repositories (e.g., for separating work and home projects). In Git, it can also be crucial for controlling the *fetch refspecs* (we will see these later, in Chapter XXX or wherever it winds up). Note that your *current repository* is defined by your current working directory, i.e., the directory as reported by the Unix/Linux `pwd` command. This will be either the top level directory of your VCS's work-tree, or a subdirectory of that. As you move from one repository to another, repository-specific settings will change automatically.

There are many more settings you can configure. We will address these later, as they come up.

Viewing

Viewing commits is just as important as creating them: if you cannot *see* what is committed, how will you see what anyone has done so far, much less plan what to do next? Here Git and Mercurial have somewhat different philosophies.

Mercurial stores each commit sequentially—locally sequentially, that is; remember from Chapter 2 that these sequential numbers are valid only within a single repository. These locally-sequential revision numbers make it easy for Mercurial to show you the entire history, starting with the highest numbered one and working backwards through the entire repository. This is what Mercurial does by default when you use `hg log`. This behavior is friendly and useful to those new to the system,[11] since it means that—unlike in Git—you will never be confused by "missing" commits.

Git, on the other hand, has just the GUIDs and graph, along with the *external references* we mentioned in Chapter 2. It has no Mercurial-like way to traverse every commit in the order (or reverse-order) it appears in the repository. In fact, there *is no* Mercurial-like order: objects within a Git repository are simply found directly by hash ID.[12]

[11] This is somewhat of a general theme: Mercurial is more friendly to new users, while Git assumes everyone starts out highly advanced. Since both systems are configurable, Mercurial's base assumption is probably superior: advanced users can configure it as needed, while new users get friendly behavior.

[12] That is, the Git repository acts as a database where each key is a hash IDs, and the value is the corresponding object.

For Git to show you *all* history, then, you must tell it to look at all references (with the `--all` option). By default, it starts instead from what it calls the `HEAD`. The `HEAD` is how Git keeps track of the *current branch*; we'll see more about this in just a moment. This also works well enough for new users: you see commits on your branch, and if you change to another branch, you see commits on that branch. However, it can be quite alarming when you get into what Git calls "detached HEAD" mode. We'll describe this mode in detail later; for now, just be aware that it can result in commit history *seeming* to disappear.

Get the repository for Git or Mercurial itself

In order to test out some of the viewing commands, you may want to download some large and complex repositories. The ones for the two VCSes themselves are useful here. We'll describe these `clone` commands in more detail later. For now, as in the xkcd comic, "just type [these shell commands] to sync up."

```
git clone git://github.com/git/git.git
```
or
```
hg clone http://www.mercurial-scm.org/hg
```

The `git clone` command creates a clone of the Git source into a new directory named `git`. The `hg clone` command creates a clone of the Mercurial source into a new directory named `hg`.

Viewing branches

The front-end commands `git branch` and `hg branches` show you *all* your branches. Git prefixes the current one with an asterisk, and both color the current branch green as well.[13] To see *just* the current branch in Mercurial, use `hg branch`. Git has no user oriented front end command to print *just* the current branch (although some of its internal commands will do it, and we'll see the `git status` command soon, which also shows you your current branch).

[13] You can configure the colors differently if you like; and as we already noted, you must enable colors in your configuration.

If we do this with the Git and Mercurial clones we just made, we will see just this:

```
* master
```

for Git, and something like this (the revision numbers will vary):

```
default                    28533:dfd5a6830ea7
stable                     28518:aa440c3d7c5d (inactive)
```

for Mercurial. The fact that Git shows only one branch while Mercurial shows two may initially be surprising,[14] but remember from Chapter 4 that Git uses *remote-tracking branches*, instead of Mercurial-style global branches. We can ask Git to show us these remote-tracking branches using `git branch -r`, producing something like this (the set of branch names may change over time):

```
origin/HEAD -> origin/master
origin/maint
origin/master
origin/next
origin/pu
origin/todo
```

We'll see later how it is that we got a `master` branch even though we have not made any commits of our own.

[14] More surprising, I think, is the fact that the Mercurial repository has just the two branches. I think this really illustrates that Mercurial's global branches are not such a great idea after all. Some public Mercurial repositories have more branches—for instance, the one for CPython has one branch per release—but nothing like Git's usual profusion.

Current branch and current revision

Both Git and Mercurial have the notion of a *current branch*, and a *current revision* or *current commit*.

The latter naturally implies the former in Mercurial: given that some commit is the current revision, and that a commit can only be on *one* branch, that branch must also be current. In fact, this principle holds in general: if you specify a revision where Mercurial needs a branch, Mercurial uses the branch containing the revision. In Mercurial, the current revision is spelled `.` (that's a period by itself), and hence `.` also refers to the current branch.

In Git, any commit may be on multiple branches at once, so "current commit" does not automatically define "current branch" this way. Nonetheless, Git still combines the two ideas. The result is that you can use the name `HEAD` to refer to either the current branch or the current revision.[15] This name takes on the desired meaning automatically. Since Git version 1.8.5 you can also use the name `@` (an unadorned at-sign). We will stick with `HEAD` here, but feel free to use `@` if you prefer (and your Git is not too old).

[15] This means you should not try to name a branch "HEAD". It is not strictly forbidden, and it will not break Git itself, but it will be confusing, like a party where all the men are named Bruce.

Git's `HEAD`

You may, of course, also refer to the current branch by its branch name. There is one exception in Git: if you are on *no* branch, the "detached HEAD" has no corresponding branch name. Let's take a moment to describe how `HEAD` works, and how detached HEAD mode differs.

In Git, the reference `HEAD` is very special. It is stored in a file in the top level of the `.git` directory. In fact, it's so special that if you

manage to delete this file, Git will stop believing that the repository is a repository.

Normally, the contents of this file are the literal word `ref:` followed by the name of the current branch, spelled out as a full external reference, such as `refs/heads/master`.[16] In what Git calls "detached HEAD" mode, however, the file contains instead a commit hash. This is quite literally the difference between being "on a branch" and not. Thus, while "detached HEAD" may sound scary, like something out of the French Revolution involving guillotines, it simply means that `HEAD` is no longer connected to a branch. This affects many other things, but for the moment, we will concentrate on commit viewing.

[16] In very old versions of Git, the `HEAD` file was actually a symbolic link to the branch's file, which was stored in `refs/heads/`. Opening the symbolic link and reading or writing its contents obtained or updated the hash. This had to be changed when Git was ported to Windows, which only supports symbolic links in NTFS.

Viewing commits

Both Git and Mercurial use the `log` command to show commits. As we already noted, when run with no options, `git log` will show commits reachable from `HEAD`. Mercurial's `hg log` will show *all* commits, in reverse order. There is an important point hiding here in plain sight: *Mercurial shows commits in reverse order. What order does Git use?*

The answer to this is a bit tricky: we already noted that there is no overall repository order. When—and *only* when—`git log` has two or more commits it *could* show next,[17] it normally *sorts them by their commit time-stamp*, with the *most recent commits* being shown first. In normal operation, this tends to be what you want, since your own commits are made in normal, Einstein-ignoring,[18] global time order: you make one commit, then you make another, and the second one is made later than the first.

Even if your own computer's clock is well behaved, Git is distributed, and you can pick up commits from other computers whose clocks are not. It's easy to pick up commits which—although they happened in one order—have time stamps that can arrange them into a *different* order. In these tough cases, Git will, by default, show them to you in the order they *claim* to have been committed. (You can also deliberately change the author and/or committer time stamps of a commit before you make it.) For instance, if your co-worker's computer puts a month-old time stamp on a new commit, that commit may be shown one month back, buried in any other commits done at that time. This is still true even if it's *your* computer that uses the month-old time stamp by mistake, though now it's *your* commit that is buried one month deep in the log. This point may seem obscure or irrelevant, until it actually happens to you and messes with your `git log` output.

[17] This condition means that in most cases, Git will show the commits in the right order anyway. At each point it is showing a commit, Git has only one commit to show. It shows that one commit and then walks to the previous commit. When you walk through merges, though, or if you attach a reference name to a future-dated commit and then use `git log --all`, the newest or most-future-dated commit comes out first.

[18] If two commits are made fast enough, or far enough apart, observers will not be able to agree which commit was actually first. More practically, Git's time stamps count in units of seconds. Commits made within a single second can sort into apparently-random order. This is not purely theoretical: a StackOverflow question [stevemao, 2015] noted that adding `--tags` changed the *order* of commits shown without changing the *set* of commits shown. This is a special case of the future-dated commit issue we just mentioned.

Graphical viewers will often show commits in strict topological order rather than in commit-time-stamp order. You can make Git do this yourself using the `--topo-order` flag. (Refer back to topological sorting in Chapter 2. Note that there may be multiple valid topological sorts. Git currently does not make any promises as to which one it will use.) Using the `--graph` option to `git log`[19] also sets the topological sort flag, in addition to telling Git to draw the commit graph.

[19] The key takeaway here is that if you find Git giving you you weird and confusing logs, using `--graph` *forces Git not to mislead you*. Any time you have a complex graph, you may want to try this option.

Mercurial, by contrast, maintains its highest-to-lowest internal number order, even when using `hg log --graph`. You can direct Mercurial to use another order by specifying which revisions to show. For instance, when using a range selector like `-r0:3`, Mercurial will show the commits in ascending order (`-r3:0` shows them in descending order).

Sample log output

Let's take a brief look at some actual commits. (I chose relatively short, recent but not too recent, ordinary non-merge commits for these.) Here is one from the Git source:

```
commit b42ca3dd0f157d0c23c9a034bc68257e1748238a
Author: Junio C Hamano <gitster@pobox.com>
Date:   Wed Oct 28 13:38:56 2015 -0700

    cat-file: read batch stream with strbuf_getline()

    It is possible to prepare a text file with a DOS editor and feed it
    as a batch command stream to the command.

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

Note that Git shows us the commit's author and date, but not the committer and commit-date. (To see those we would need to specify a different log output format.) We also see the commit's *log message*, indented by four spaces. The log message has the form of a short subject line, followed by a blank line, followed by a longer description of what the commit does (and in this case, ended with one or more "Signed-off-by" lines, which Git can add automatically; not all projects use this feature).

Here is a similar commit from the Mercurial source:[20]

[20] Both of these commits have good commit messages. We'll examine just what makes them "good" in more detail later.

```
changeset:   27373:84784f834b3a
user:        Gregory Szorc <gregory.szorc@gmail.com>
date:        Sun Dec 13 11:27:52 2015 -0800
summary:     help: add documentation for bundle types
```

The format is essentially the same as Git's, except that by default, Mercurial shows only the one-line summary (subject). Examining the

same commit in more detail—we'll see just *how* to do this later—we would find the full commit description to read:

```
help: add documentation for bundle types

Bundle types and the high-level data format of each bundle isn't
documented anywhere. Let's document this as well.

Obviously there are many more details about bundles that could be
written about. But you have to start somewhere.
```

For now, note how similar these outputs are: we see the commit's ID, the commit's author and date, and a log message.

Limiting or augmenting the commits shown

What if, in Git, you want to see *more* branches? The `log` command accepts branch names as arguments. If you give at least one, Git will *not* start from `HEAD`, but only from the branch or branches you give as arguments.[21] For instance, when looking at the Git source repository we just cloned, using `git log origin/next origin/maint` will ignore `HEAD` and show us commits reachable from `origin/next` and `origin/maint` instead. As before, these will be sorted in commit timestamp order unless we direct Git otherwise.

What if, in Mercurial, you want to see *fewer* branches? In particular, you might very often want Git's behavior of showing you just your current branch, rather than every branch. In this case, you must add a specifier that selects the desired branch. Since `.` is the current revision (which is on the current branch), and `hg log -b` takes a branch name, `hg log -b .` does the trick. (You can also spell this `hg log -r 'branch(.)'` but this sorts the commits starting from revision `0`, so that you need `hg log -r 'reverse(branch(.))'` instead.)

[21] This is wrong in a technical, nitty-detail way: `git log` first translates from name to commit hash, then selects that commit with ancestry so as to walk through the commit graph, hence showing the branch. That is, `git log` doesn't "start from the branch" at all, but rather from the *branch tip*. We covered this back in Chapter 2, but it's worth repeating, and we will see it all yet again soon.

Viewing with a detached HEAD

As we just saw, `git log` starts from `HEAD` by default. Suppose that you are on branch `master`, and you decide you need to look at the code the way it was a month or more ago. You can use the command `git log` to find an old commit—we saw one just recently: `b42ca3d...` in the Git source—and then check it out, using `git checkout b42ca3d` (you can abbreviate these hash values). Git prints out a large warning beginning with:

```
Note: checking out 'b42ca3d'.

You are in 'detached HEAD' state. ...
```

Now that `HEAD` points directly to commit `b42ca3d`, `git log` shows commits working back from October 28, 2015. All the newer commits *seem* to have vanished! No worries, though: `git checkout master` brings them all back into view, or of course we could use `git log master` to see them.

6
Getting started

We are finally ready to get started in both Git and Mercurial. We will examine two different startup scenarios: creating a *new* project, and cloning an *existing* project that we will work on and contribute to. Along the way we'll accidentally (or not so accidentally) make some mistakes, just to show how to recover from them.

This is also where Git and Mercurial first begin to diverge in the way they are used. We will cover both here, but we will see some differences immediately.

By the end of the chapter, you will be able to create and/or clone a project, make new commits in a repository, fix silly mistakes in any just made-commit, and detect whether you *need* to make a new commit. You will be able to create a new branch, and to switch from one branch to another. For Git, you will learn the special—and slightly crazy—way it deals with the fact that branch names are *not* global and permanent, as we saw back in Chapter 4. You will learn some of the tricks needed to identify *specific* commits, namely those that are not branch tips (Git) or heads (Mercurial). Perhaps most importantly, you should begin to understand how and why the commit DAG grows, accumulating new history, as you make new commits.

New projects: create, commit, and view commits

Both Git and Mercurial initialize new projects with the `init` verb, which should be run in a freshly created directory:

```
mkdir project
cd project
```
then one of these:
```
git init
hg init
```

Either one creates a new repository[1] and sets things up so that your first commit will create the VCS's default branch: `master` in Git, or `default` in Mercurial. (You must pick one particular VCS now,

[1] Git's `init` can be used in an existing repository. This is harmless—it won't touch any existing work—but not useful to us yet. Mercurial's `init` refuses to do anything in an existing repository, so is also harmless.

or at least, one per project—you could make a `project.git` and a `project.hg`, for instance.)

With both systems, you now create whatever files you want to have in the first commit. Including a file named `README` is often a good idea, so let's create one now, with something in it.[2]

```
echo Marsupial Madness > README
```

(If you prefer, create the `README` in your editor, or any other way.)

Next, you must explicitly `add` this file, so that the VCS knows it should include it in the next commit. Here Git and Mercurial differ: Git requires you to `git add` every file every time you want it to be updated with the next commit, while Mercurial needs just one `hg add`, the first time you create the file. After that, if you have edited the file, the next Mercurial commit will automatically include any changes you made to the file. Which method is "better" is a matter of taste, and the Git and Mercurial authors both consider this difference a feature in their favor.[3] For now, since `README` is new, it makes no real difference. Use the obvious one of:

```
git add README
hg add README
```

We are now ready to make our first commit, which will create the branch:

```
git commit
hg commit
```

This will bring up your editor, specifically the one you selected with `core.editor` or `ui.editor`. It is your job now to enter a good commit message, write out the file, and exit the editor.[4] We'll see how to write a good commit message soon. For now, you might as well use something like "initial commit" or even just "initial," which you can simply type in as a single line. Write out the file and exit your editor to get the first commit made. (Mercurial does this silently; Git prints something to confirm your new commit has occurred.)

Now let's take a look at the (single) commit. We did this in Chapter 5, but let's look a bit closer.

```
git log
hg log
```

With Git, the output should resemble:

```
commit 5318e618785487817de1803a4395853511ee78d5
Author: Chris Torek <chris.torek@example.com>
Date:   Wed Apr 19 02:45:29 2017 -0700
    initial commit
```

[2] The Git repository sharing web site github.com suggests naming the file `README.md`, where the `md` suffix indicates that the file's contents use *markdown* syntax (markdown syntax is outside the scope of this book). Github will present these contents on the main web page for the project.

[3] We will see *why* the two systems behave like this, and how to control it in more detail, in Chapter 7. Either system *allows* the other one's behavior, so both systems are in fact equally capable.

[4] Some editors, such as some varieties of atom, emacs, and sublime, may require you to run a special command or use special options here. If you run the editor directly from a shell, it may act as an *proxy agent* for a window-based variant of the editor. The window-based editor then opens the file for editing, and the agent immediately exits with a success indication. Both VCSes assume the commit message file is complete as soon as the sub-process they spin off also completes. Hence, instead of running the proxy agent that exits too soon, you must run one that waits for a "file is done" signal. Consult your editor's documentation for details.

and with Mercurial, it should look like this:

```
changeset:   0:1d84a50ae05f
tag:         tip
user:        Chris Torek <chris.torek@example.com>
date:        Wed Apr 19 02:46:21 2017 -0700
summary:     initial commit
```

Git shows you the full hash while Mercurial shows you the locally-sequential revision number and an abbreviated hash. We only entered the one subject line in the commit message, so the difference in the log message output essentially vanishes. Finally, Mercurial includes this slightly mysterious "tag" line; we'll get to this later.

Let's create one more revision so that we have two revisions to look at in the main branch.

```
echo "kangaroos are large and grey" > kanga.txt
git add kanga.txt
git commit -m "add a kangaroo"
```

or

```
echo "kangaroos are large and grey" > kanga.txt
hg add kanga.txt
hg commit -m "add a kangaroo"
```

Since `kanga.txt` is new, we still have to `add` it in both VCSes. This time, though, we used the `-m` switch, which takes a commit message. This skips the editor session, at the cost of limiting you to a one-line commit message.[5]

[5] You can supply more than one line here, but the methods are a bit command-line-interpreter dependent, whereas the method shown here should work with any standard CLI.

While Mercurial is quiet when adding the new file, Git prints out a bit of information at the end:

```
[master 3c345b0] add a kangaroo
 1 file changed, 1 insertion(+)
 create mode 100644 kanga.txt
```

The first line gives the branch name and an abbreviated version of the new commit's hash, and the one-line summary log message. The remaining lines give a summary of what changed from this commit's parent commit. (If we had added more files, we would get more "create mode . . . " lines.)

Let's take a look at the logs now. This time, let's direct Git to produce the one-line summary, rather than the full message:

```
git log --oneline
hg log
```

The Git output is now:

```
3c345b0 add a kangaroo
5318e61 initial commit
```

Note that Git's one-line summary format is much more abbreviated than Mercurial's default. Meanwhile, Mercurial lacks a one line format.[6] It does have a verbose log mode, which prints the entire commit message instead of the one line summary, but there is no point in using it yet.

[6] We can obtain one through what Mercurial calls *templates*.

```
changeset:   1:d05b1df8b8f6
tag:         tip
user:        Chris Torek <chris.torek@example.com>
date:        Wed Apr 19 02:49:21 2017 -0700
summary:     add a kangaroo

changeset:   0:1d84a50ae05f
user:        Chris Torek <chris.torek@example.com>
date:        Wed Apr 19 02:46:21 2017 -0700
summary:     initial commit
```

Now let's make a new branch *starting from the initial revision*, so that we have one commit on a side branch, parallel to the second commit on the main branch. We'll see some contrasts between Git and Mercurial here. Our first task is to get back to that initial revision, because both VCSes default to making a new branch that starts from whatever the *current revision* is.

Switching revisions

Git and Mercurial have similar methods of keeping track of your current revision. Both of them also use a `checkout` command to check out a specific revision, although Mercurial users usually call this `update`. (The verbs mean exactly the same thing in Mercurial: unlike some other VCSes, `update` and `checkout` are simply aliases for each other. Mercurial even gives you a *third* name for this verb, `co`. Git has just the one verb initially, although you can define as many aliases as you like.)

When we want to check out one particular commit, the most direct way is to name it by its raw hash ID. Both Git and Mercurial support this. Because the hash is the GUID of that commit, it *always* works to identify *exactly that commit*, no matter where this commit is in the commit DAG. The drawback is that you must type in apparently-random numbers. The good news is that you can shorten the hash: instead of `5318e618785487817de1803a4395853511ee78d5`, you can type in `5318e61`,[7] but really this is not much of an improvement. I find that it works well to cut-and-paste these, but otherwise I generally try to find an alternative.

[7] The shortest Git allows is four characters, while Mercurial allows even fewer, but whatever you type in must match exactly one actual hash in the repository. A four character abbreviation is not as likely to be unique as a seven or eight character abbreviation.

Git has many alternatives—far too many to list them all just yet. We'll restrict ourselves to just one for now, specifically, the syntax that means *move back one parent in the DAG*. To do that, we name any

revision any way we like, and then append a caret or hat character.[8] Since `5318e61` is the parent of `3c345b0`, writing `3c345b0^` is another way to write `5318e61`.

This hardly seems helpful—we've merely substituted one incomprehensible hash for another. But as we saw in Chapter 5, page 89, the *current* revision is called `HEAD` in Git. Hence, all we need to do to "back up one commit" is to write `HEAD^`.

Mercurial, of course, uses `.` instead of `HEAD`. Mercurial also has those more-convenient (and initially *much* shorter) sequential numbers: the first commit is revision 0, and the second is revision 1. Since we have just the two commits now, we know we want revision 0—but doing this is, in a sense, cheating. In a real code base that has a lot of development, we won't know which revision number to use.[9] In that sense, it's better to use DAG-following operators, just as in Git.

As luck would have it, both VCSes use the *same* syntax for walking back through the commit DAG. This means we can use the obvious one of these two commands to check out the initial commit:

```
git checkout HEAD^
hg update -r .^
```

Since `HEAD` or `.` is the current commit, and suffix-`^` means *parent*,[10] this steps back one commit in the DAG.

Git: switching revisions

Let's run the appropriate the command and observe the output. Git's is a bit scary:

```
Note: checking out 'HEAD^'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 5318e61... initial commit
```

If we run `git log` (try it now), we now see only the first commit. Where did our second commit go?

This is just what we mentioned in Chapter 5: `git log` *means* `git log HEAD`, and `git log` starts from the commit you identify, then walks backwards—never forwards—through the commit DAG to find commits.[11] Then it shows you these commits, one at a time, in some

[8] On Windows, some command line interpreters steal the `^` character for their own use. If you are stuck with this, you can use `~1` instead: the tilde character followed by the digit 1. We'll fill in more details for this syntax later.

[9] It's tempting to run `hg id -n -r .` to find the current revision number, then subtract `1`, but this eventually runs into issues with complex DAGs.

[10] If the commit in question is a merge commit—remember that merge commits have two or more parents in the DAG—it means to use the commit's *first* parent. We'll see more about this when we get to merging.

[11] In other words, `log` selects the specified commits *with ancestry*, just as we saw in Chapter 2, page 43, and again in Chapter 5.

order. Since we moved `HEAD` back to the root commit, `git log` now has just one commit to show.

This does make it quite easy to see that we are now on the root commit. If you want to see more commits, not just those starting from the current revision, you have to give more arguments or options to `git log`. For instance, try `git log master` now. However, starting from `HEAD` is often what you will will want.

Meanwhile, the "detached HEAD" message is just telling us that we are no longer on *any* branch.[12] The branch-name `master` still exists but we are now off that branch, and anything we commit now will eventually be thrown away unless we get back on a branch that will keep it. This is, of course, the plan: we want to get on a new branch and make a new commit, and we want the new commit's parent to be the root commit, where we now have this detached HEAD.

[12] You could instead say that we are on Git's single special anonymous branch. The Git documentation is reasonably consistent about saying that we are not on any branch, but internally, we're just on a branch whose only useable name is `HEAD` or `@`.

While we are in this mode, let's look at two more Git commands. First, run `git status` and observe the output (your commit GUID will differ, if it appears at all):

```
HEAD detached at 5318e61
nothing to commit, working directory clean
```

The exact phrasing depends on your Git version: before version 1.8.3, the first line would just read `Not currently on any branch.`

Remember `git status`: it is a *very* useful command, especially in versions of Git since 1.8.2. It improved significantly in 1.8.3, 1.8.4, and 1.8.5 and has had minor improvements since then as well. If you are ever in the middle of some operation, and are no longer sure as to what is going on,[13] `git status` should tell you where you are and remind you how to continue, or if you prefer, terminate the operation.

[13] With the commands we have used so far, things either work, or completely fail, but when we get to merge and rebase and the like, many commands can stop, return you to the CLI, and wait for instructions. You may then get interrupted at work and forget what you were in the middle of, for instance. Use `git status`!

Now run `git branch`. The output will be something like this:

```
* (HEAD detached at 5318e61)
  master
```

Again, the details will vary based on version, but the main thing is that the starred line tells us which branch we're on, or in this case, not on.

Mercurial: switching revisions

Compare all this with Mercurial's reaction to backing up one step in the commit DAG:

```
0 files updated, 0 files merged, 1 files removed, 0 files unresolved
```

There is no scary message that sounds like the French Revolution is underway and our head is in the guillotine. And in fact, we are still on the standard `default` branch, which we can see with `hg branch`. Let's run that now and observe the output:

```
default
```

Mercurial has the same `status` command as Git, but it's remarkably uninteresting right now as it prints nothing at all. We can also run `hg log`. Please do that now; but then observe that it prints exactly the same thing it printed before, which is not terribly helpful. If we want to know where we are now, we need a different command, or maybe an option. There are two commands that will do the trick: `hg id` and `hg summary`. Let's try the first one first:

```
1d84a50ae05f
```

This is the abbreviated hash for our first commit (your string will therefore be different, but look at the `hg log` output and note that the hash matches that of the first commit). We can use `hg id -n` instead: this prints 0, which is the sequential number assigned to the first commit (the 0 part of `0:1d84a50ae05f`). Let's try `hg summary` instead:

```
parent: 0:1d84a50ae05f
 initial commit
branch: default
commit: (clean)
update: 1 new changesets (update)
phases: 2 draft
```

This prints out a lot more useful stuff. The first line, although it says "parent", is the current revision we have checked-out right now, which is just what we wanted to know. The second line is the one-line commit log summary, and the third is the current branch. We can ignore the last several lines for now.

The other trick we can use in Mercurial is to direct `hg log` to show us just the current commit, using `hg log -r .` to specify which revision to show. This is what I actually use most of the time, typically with `-v` as well, so that I can see the full log message. Unlike Git, Mercurial's `log` does not automatically walk back through the DAG; if we want to show more revisions, we need to use a DAG range selector such as `hg log -r 'ancestors(.)'` or `hg log -r ::.`. You might recall these operations from Chapter 2, page 45. If not, you may want to review that chapter soon.

Creating a new branch

We're about ready to create the new branch, but before we do, take a look at your working directory. Note that the file `kanga.txt` has disappeared. This is because you told the VCS to step back from the branch-tip commit (where we added the file) to the initial commit (where we did not have it).

Making the new branch is easy now that we're on the desired (initial) revision. We just need to ask the VCS to change to the new branch, then make a new commit on that branch. In Git, in fact, we don't even need to make a new commit, because the existing initial commit can be on two or more branches, but in Mercurial, this is not allowed. In any case, let's create the branch, then make a new commit.

Git: creating a new branch

In Git, the `branch` command can create a new branch, but it does not automatically *switch to* that branch. We could use two separate commands:

```
git branch sidebr
git checkout sidebr
```

but we can, and might as well, combine them into one:

```
git checkout -b newbr    oops, wrong name!
```

The `-b` option to `checkout` tells it to create and switch to the branch, all at one go.[14] The branch creation step uses the current, i.e., `HEAD`, commit as the tip of the new branch, and the switch-to-branch action of `git checkout` puts us back on that branch (so that our head is firmly reattached to our shoulders). Let's use the combined command and observe the output:

```
Switched to a new branch 'newbr'
```

Oops! I meant to name this branch `sidebr`, but I accidentally entered `git checkout -b newbr`. Fortunately, there is no problem here. I can either re-do the checkout with the correct branch name (which will leave me with an extra branch that I will have to delete later), or use `git branch -m sidebr` to change the name of the current branch to `sidebr`. Note that `git branch` with no options will list all our (local) branch names; try it before and after fixing the branch name:

```
git branch
git branch -m sidebr
git branch
```

[14] This, as it turns out, is a general theme in Git: someone provides a command to do something that doesn't quite do enough, and then there is either an additional command added, or more likely an option added to an existing command, to do everything together. The result is that many commands do too much, making it easy to make mistakes. Fortunately Git makes it easy to undo mistakes. Mercurial typically makes it much harder to make mistakes, but often much harder to undo them as well.

Mercurial: creating a new branch

Let's do the same in Mercurial, which also uses `hg branch` to change the branch name:

```
hg branch newbr
```

The output this time is a little scary:

```
marked working directory as branch newbr
(branches are permanent and global, did you want a bookmark?)
```

Bookmarks are Mercurial's answer to the global nature of its branches. For now, we'll ignore them since they make Mercurial's usage more complicated—in fact, more like Git's. I made the same mistake again though, naming the new branch `newbr` instead of `sidebr`. Fortunately, branches in Mercurial *cannot* exist without a commit on them, so to fix this, I just have to re-run the `hg branch` command with the correct name. We can use the command `hg branches` to list all the branches[15] Mercurial has in this repository, and we will see that neither `newbr` nor `sidebr` exist yet:

[15] Well, all the *open* branches, but let's ignore this complication for now.

```
hg branch sidebr
hg branches
```

(you will have to run these yourself to see the output).

Both: making changes on the side branch

In Git, our new branch already exists now, pointing to the initial commit. In Mercurial, it doesn't exist yet, but our current revision *is* the initial commit. In either case, let's make a new commit now. Instead of creating `kanga.txt`, though, let's modify the `README` and add a koala.

```
echo Add a line to README >> README
echo koalas look cute and cuddly > koala.txt
```

(Again, you can use an editor to modify the `README` and add the new `koala.txt` file, if you prefer.)

Git's status versus Mercurial's

Now let's run the VCS's `status` command. We have reached a point where Git and Mercurial differ again, and this difference is actually helpful and instructive.

The `git status` output reads:

```
On branch sidebr
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:   README

Untracked files:
  (use "git add <file>..." to include in what will be committed)

koala.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

This is pretty verbose and is full of new concepts,[16] but let's compare it to what Mercurial says, because both VCSes really do very similar things. The `hg status` output reads:

[16] We'll go into more detail in Chapter 7. For now, let's just concentrate on getting our changes committed.

```
M README
? koala.txt
```

This is much shorter than the Git output, but in fact, it has pretty much the same information, and we can make Git use short output too, using `git status --short`:

```
 M README
?? koala.txt
```

Note that Git's short status is still suspiciously different from Mercurial's: each file is listed with the same characters, but in Git's output, the question-mark is doubled. The uppercase M character, which stands for "modified" and appears in front of `README`, has a leading space in Git's status, but not in Mercurial's.

This all goes back to the note we made earlier where Git requires us to `git add` each file before every commit. Both Git and Mercurial mark the new `koala.txt` with a question mark (or two), meaning they know nothing about this file.

When we `add` a file in Mercurial, this adds the file to what Mercurial calls its *manifest*. This is just a fancy word meaning *list*: in this case, a list of all files Mercurial should look at when making a new commit. The `hg status` command compares all the listed files to their work-tree version, and if they are changed, `hg commit` will include their changes in the next changeset (commit).

In Git, there is no manifest. Instead, Git provides a more complex construct called the *index* or *staging area*.[17] In any case, you must use `git add` to update each file into the staging area before every commit. It is tremendously easy to forget one, so we will do that now, and then fix it. To make things more interesting, let's make this mistake in *both* VCSes.

[17] The reason for multiple names is mainly historical. The name *index* is simply the older, original name, while *staging area* is meant to describe it better. However, the index plays multiple roles, and sometimes "staging area" area is misleading as well. We will come back to this idea repeatedly. It is considered a feature in Git, as it allows for alternative work-flows, but it is highly intrusive as well: Git forces you to be aware of it. For now, just note that Git has it, and Mercurial doesn't. We'll go into much more detail in Chapter 7.

Committing a mistake: getting our `add` *wrong*

In Git, let's forget to add the *new* file, just adding the changes to the `README` file:

```
git add README
git commit
```

When this brings up the editor, the file into which we are to write a commit log message looks like this:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch sidebr
# Changes to be committed:
#       modified:   README
#
# Untracked files:
#       koala.txt
#
```

This is essentially just the same `git status` output we saw earlier, except this time `README` is included in what is to be committed.

As the instructions say, we can simply exit the editor without making a commit at all. Then we can `git add koala.txt` and restart the editor. Let's make another mistake and put something in the file, though, so that we get a commit. While we're at it, let's come up with a better commit message.

Before we go on, let's see this same process with Mercurial, i.e., forgetting to add `koala.txt` and simply running `hg commit`. When this brings up the editor, the file looks like this:

```
HG: Enter commit message.  Lines beginning with 'HG:' are removed.
HG: Leave message empty to abort commit.
HG: --
HG: user: Chris Torek <chris.torek@example.com>
HG: branch 'sidebr'
HG: changed README
```

The forgotten file is not listed this time, so it's easy to miss. If you do catch it, Git and Mercurial use the same technique where an empty message stops the commit from happening. For now, though, we're going to try to write a good commit message, and go ahead and do the wrong commit.

A good commit message

To some extent, what makes a commit log message good or bad is a value judgement that cannot be reduced to simple rules. Nonetheless, if we consider the *context* in which commit messages will be

viewed later, a number of simple options become clear. Both Git and Mercurial offer both one-line and verbose (multi-line) views of log messages,[18] and both also have ways to search through commit log messages for particular strings and patterns. You or one of your collaborators will view these messages later, with an eye toward finding where a bug crept in or was fixed, whether a feature is finished yet, or quite often, what the *purpose* of some particular change was.

[18] As we noted earlier, in Mercurial, the one-line message requires writing a template.

```
* 80a7ae7 switch cli env to quoting package
| * 858277f (obeylines-exp) experiment with \obeylines
|/
* 0c4101d add cli environment, use in "about"
```

Figure 6.1: Fragment of `git log --oneline --decorate --all` output.

Many viewers will try to present you with an abbreviated commit ID, some graphical drawings connecting parent and child commit nodes with lines, possibly some branch and/or tag names and so on, *and* one line—the *subject* line—taken from the commit message, as shown in Figure 6.1. This means that the first line of a log message should be short and punchy, based on an action. A good rule of thumb is that it completes the sentence: *If you accept this change, it will* .... A commit to `roo.txt` might therefore begin with *chase out a wallaby*. This might be prefixed with a subsystem or file name (although file names are easy to extract from commits).

This short, punchy, action-oriented *verb the object* style subject line should be followed by a blank line. This is how Git and Mercurial know where the one-line message stops (Mercurial will just stop after the first log message line anyway, but Git will keep assembling more text until it comes across a blank line). Below the blank line, write as much additional text as you need to remind the future version of yourself, or explain to your collaborators, what you were thinking when you made whatever changes you were working on. I also recommend keeping these lines not too long (maybe up to 70 or so characters, as they will get indented by various commands), and using a blank line between paragraphs.

In this particular case, the commit message I used is:

```
add prototype koala file

Prepare for working on koalas in side branch.
While we're at it, update the README.
```

which actually highlights a bit of bad practice: the "while we're at it" part. A good commit *message* describes a single action, and "update the README" is an action that seems (and in fact is) unrelated to any koala preparation. Commits themselves should also only do one thing, as much as possible anyway, since we can, in the future, look

at these changes in isolation (to see if bugs, or perhaps wallabies, got in) or actually back them out (if necessary). It's not so good if backing out a bug also backs out some unrelated wording or spelling fix in some documentation. Ultimately, this is still a good commit *message*; it's just describing a bad *commit*.[19] If what we changed in the `README` file had been properly koala-related, this would be a good commit. Still, let's write it out and exit the editor, so that the VCS adds our third commit.

[19] Ironically, our "mistake"—failing to add `koala.txt`—would actually make it easy to make *two* commits now, one just for the `README` and then one just for the new file. That would actually be a better idea, but *forgot to add a file* is a very common mistake so let's run with this example anyway.

Fixing an incorrect commit

In any case, this commit has a bigger mistake: we forgot to add the `koala.txt` file. Let's see how to fix this, which is easy enough in both Git and Mercurial, since it's just one commit. It's important to note, though, that *we have not published this commit*. No one else has it, which is what allows us to fix it.[20]

[20] Incidentally, this is why allowing colleages to fetch or pull at any time, without warning, from your private repository is generally a bad idea. You don't want them to get commits that are not yet ready. Mercurial's *commit phases* help here—we will describe these later—but it is generally just not a good plan.

Git: fixing an incorrect commit

In Git, to fix a previous commit, we use `git commit --amend`. This redoes the commit, allowing us to edit the commit log message. Before we amend the commit, though, we want to stage (i.e., add) `koala.txt` this time. (If we forget again, we can just keep doing the amend.) Hence:

```
git add koala.txt
git commit --amend
```

We don't have to re-stage `README` here, since it's already staged from earlier.[21] You can think of this as arranging the furniture and props on the kind of stage used in a play, and then taking a snapshot: the picture saves the staged arrangement, but it's still staged.

[21] If you're not sure what's staged, just run `git status`. Still, it won't hurt to re-stage it, either.

If we *did* need to fix something in `README`, we could edit the file in the work-tree, re-stage it with `git add README`, and do another `git commit --amend`. You can run as many amends as you like, one after the other. Each just hides away the previous commit while leaving the stage alone, then makes a new commit snapshot that takes the place of the earlier one.

Mercurial: fixing an incorrect commit

In Mercurial, to fix a commit, we could first explicitly *roll back* the one we just made, using the command `hg rollback`:

```
repository tip rolled back to revision 1 (undo commit)
working directory now based on revision 0
```

Now we can `hg add koala.c` and run `hg commit` again. Unfortunately, this does not save the original commit message, so we would have to type it in all over again. Mercurial therefore acquired a `commit --amend` option in version 2.2. This effectively combines the rollback-and-recommit in much the same way as Git's `commit --amend` (though under the hood, there's no more need for the rollback step, which actually makes it *more* useful).[22]

Instead of the three step rollback, add, re-commit, you can and should use the simpler sequence:

```
hg add koala.c
hg commit --amend
```

which works identically to the method in Git.

[22] Furthermore, the use of `hg rollback` is discouraged in Mercurial since version 2.2. This is because rollback is implemented at the wrong level: it undoes the last *internal database transaction*, not necessarily the last *commit*.

Both: fixing an incorrect commit

The one good thing about Mercurial's old (pre-version-2.2) method for fixing a commit is that it makes it clear how this actually works. Remember that we saw in Chapter 4 that the GUID of a commit depends on every part of its contents. This means that it is literally impossible to *change* a commit. All we can do—which is thus precisely what we *do* do—is to make a *new* commit, with new contents and new log message, whose parent commit is the *same* as the original parent. There is a significant difference between the two VCS's implementations here, though. When using `hg commit --amend`, you will see a message like this:

```
saved backup bundle to ... c9974a6107c4-e05b2a02-amend-backup.hg
```

This "backup bundle" holds a copy of the commit that has been removed from the repository. (The path name of the backup is in the repository directory, but these are at least logically separate; they have to be *unbundled* to turn them back into commits.) Mercurial must remove this commit because otherwise it would present itself as a new head within the branch. Git leaves the old commit behind in the repository, while simply modifying the branch name to point to the new commit. In other words, Git *leaves the original commit in the commit DAG*, which means you can get to it again as long as you do so before it expires. Admittedly, if you want the original commit back, finding it can be a bit tricky, somewhat analagous to searching through the backup bundles Mercurial saves, but there is no need to transform ("unbundle") them: all the usual Git operations work normally on these commits.

How `HEAD` works and branches grow in Git

Branch growth in Mercurial is easy: `hg commit` just makes a new commit on the current branch, setting the new commit's parent ID to the current commit ID, and then setting the current commit ID to the newly-made commit. The new commit is automatically a Mercurial head within the branch since it has no commits pointing to it yet. If it is the only head, everything is *very* simple, but even if not, the *process* is simple and we can deal with the multiple heads later. In Git, though, a commit may be on many branches, or even no branch. How can this actually work?

Exercise 6.1: When is a new commit the *only* head, and when is *another* head?

We noted earlier that in Git, the name `HEAD` always identifies the current commit. We also just saw that checking out a commit by ID, or by relative name like `HEAD^`, "detaches" `HEAD` as if we were some ghoul chasing Ichabod Crane. Creating or getting back on a branch using `git checkout` somehow re-attaches our `HEAD`. What's actually going on here?

As we noted in Chapter 5, Git's `HEAD` normally contains the *name* of a branch—or slightly more precisely, the *name of a branch name*. Git calls this a *symbolic reference*.[23] When `HEAD` has another branch name inside it, Git says that we are on that branch. This affects the way `git commit` makes new commits.

[23] You might wonder if Git allows other names to be symbolic references. It does; but they are not actually very useful. Git decides when to follow a symbolic reference through its target (vs using it symbolically) mainly by hard-coding the correct special-case action only for `HEAD`.

Remember that we say that a branch name like `master` "points to" the tip commit of branch `master`. This means that Git's branch-table entry for `master` contains the raw hash ID of that commit.[24] To make a new commit, the `git commit` command reads `HEAD`, sees that it says `master`, reads `master`, and finds the current commit ID. It then writes that commit ID into the new commit as its parent-commit ID. Once the new commit is safely ensconced in the repository, `git commit` writes the *new* commit's ID back into the entry for `master`. `HEAD` still points to (i.e., names) `master`, but `master` now points to the new commit. The new commit points back to the previous tip of `master`, and we have successfully added a commit to `master`, as in Figure 6.2. The old, now-overwritten commit hash that was stored in `master` is represented by the dashed grey line; the new hash, in solid blue, points to the new (solid blue) commit.

[24] More precisely, the `refs/heads/master` entry has the ID. This entry may be in a *packed* refs file or in its own separate file. Git's abstractions for working with these have improved greatly since the old days, and programmers should no longer peek directly at these files.

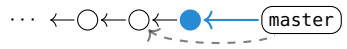

Figure 6.2: Adding new (blue) commit.

To detach `HEAD`, Git simply writes an actual hash ID into it, instead of the name of a branch. Now `git commit` reads `HEAD` and, since that resolves to the hash ID, stops there. It then makes the new commit as before but writes the new ID directly into `HEAD`. The anonymous branch therefore grows as we make new commits. Using `git checkout -b` *`newbranch`* to create a new branch name copies the hash ID from the detached `HEAD` to the new branch name entry, then writes the branch name into `HEAD`. (If this whole paragraph make no

sense to you, don't worry! We'll come back to it later.)

Cloning existing projects

Starting with an existing project is in one sense easier than creating a new project: you get a whole bunch of commits you can look at and play with, without having to write them yourself. In fact, we did this in Chapter 5, and it took just one command.

There is, however, a great deal going on with all of this. Cloning immediately exposes you to some major differences between Git and Mercurial. The Mercurial startup process is quite straightforward, but the Git startup process is not. It is easy to *do*, and it *seems* simple, but without proper preparation, you may soon go far astray and end up deleting the project and downloading a fresh copy.

Moreover, once you choose to send your commits *back*, or store them on a cloud server such as GitHub or KilnHg, you will need to *authenticate*: to prove to the server that you are who you claim to be. We'll touch lightly on the issue of authentication here, though the details depend too much on both the server and your own OS to show everything.

The first step is the same for both VCSes; you will clone the existing project from some URL:

```
git clone url
hg clone url
```

The general form of a URL should be familar to anyone who has used a browser: `http://host.dom.ain/path/to/thing`. The first part of this URL ("`http`") is called a *scheme*. Both Git and Mercurial have four built-in schemes: a local "`file`" path, an "`http`" or "`https`" web site, or an "`ssh`" host and path. Using an absolute path without a `file` scheme will direct both VCSes to a local file as well.[25] Git supports one more scheme than Mercurial, using "`git`".[26]

In general, these URLs use a double slash followed by an optional user name (and even an optional password[27]), then the host name, an optional port, and then a path with slashes:

```
scheme://host/path/to/repository
scheme://user@host/path/to/repository
scheme://user@host:port/path...
```

These work with both `https` and `ssh`. Git's `git:` scheme does not take a user name and does no authentication, but does allow a non-standard port, as in `git://host:port/path...`.

The scheme you choose selects how the client will talk to the server, specifically, a *protocol* and *transport layer* by which client and server can converse. In general, I recommend using `ssh` or `https`

[25] Both VCSes support Unix-style paths with leading slashes. Windows-style paths using backslashes can be trickier, and Windows drive-letters look like scheme prefixes. Some versions of Git-for-Windows handle these better than others.

[26] Git actually has five more: `ftp`, `ftps`, and `rsync`, and also `git+ssh` and `ssh+git`. The last two are just alternative syntaxes for the `ssh` scheme. The other three are deprecated and may be removed without notice.

[27] Showing passwords in cleartext is a bad idea; this will not be done here.

if possible, as once it they are set up, they provide a well-secured, well-authenticated transport layer.[28] For ssh, you generate a secure key once on your client, and then install it on the server, in some server-specific manner. With https, you must supply a user name and password somehow; the SSL/TLS session encrypts these when sending them to the server so that only the server can decrypt them.

If you must provide a password frequently, you will want some sort of *credential helper* program, and/or an *agent*. Git includes several built in credential helpers for https authentication, but you will probably want a system-specific one such as OS X Keychain. If your ssh keys are protected by a passphrase, either OS X Keychain or ssh's `ssh-agent` command can authorize programs to access the keys. Since these details are OS-dependent, they are largely outside the scope of this book.

Once you get past the transport layer, neither Git nor Mercurial have any built-in verification that users are who they claim to be (the name and email-address settings are simply passed on through). If you are simply cloning a *public* repository, of course, any of the non-authenticated methods are fine as well.

Both Git and Mercurial will perform the clone as directed, copying the remote's repository to a new local repository whose name is the same as the final path component. For instance, if you clone `git://github.com/git` or `http://github.com/git` you get the source code to Git in a directory named `./git`; if you clone `http://www.mercurial-scm.org/hg` you get the source code to Mercurial in a directory named `./hg`. With both Git and Mercurial, cloning also saves the original URL in the new clone's configuration file. However, since they use different methods for distributing branches, the Git clone will have only one *local* branch, while the Mercurial clone will have *all* branches. As we just mentioned, the Mercurial setup is much simpler: at this point you are ready to go. Git's is not, so let's examine it closely.

[28] Https uses SSL/TLS, while ssh—which stands for Secure SHell—is similar but has its own protocol. SSL stands for Secure Socket Layer, and TLS for Transport Layer Security. SSL/TLS set-up is especially complex on the server, which must do a lot of certificate authentication. As a user simply downloading files, few of these setup issues affect you.

Git: cloning existing projects

Remember that in Chapter 4, we noted that Git uses a *remote name* to qualify *remote-tracking names*. When you clone any repository, the name of the remote is `origin`.[29] This is meant to show that everything so recorded came from the "origin-al" repository you just cloned. If you intend to send changes back, some people (and some parts of Git's own documentation, for that matter) prefer to call this the *upstream*, so you will see both of these names at times.[30] In any case, the Git clone will copy all of the remote's branches—by this we mean the *ordinary branch names* as they appear *on the remote*—into

[29] You can change this at clone time, or at any time later, if you like.

[30] There is also a more complicated setup you can use where you will have both an `origin` *and* an `upstream`, but we will leave that for later.

your *remote-tracking names*, such as `origin/master`.

As soon as the clone is ready to use, though, Git creates one local branch name for you. Usually this is named `master` (but we'll see a different branch name used in just a moment). Since each Git branch name points to that branch's tip commit, the question you should ask yourself here is this: *How does Git decide which commit your newly-created local branch is to set as its tip?* The answer to this question is both surprisingly simple *and* surprisingly complicated, and lies at the heart of the first stumbling block for many Git users: it's the *same* commit as that of a remote-tracking name.

The last step that `git clone` runs is `git checkout` *`branchname`*, such as `git checkout master`.[31] Normally this command means *switch me to that branch*, but the branch does not exist yet. What `checkout` does here is to see if there is one (and *only* one) remote-tracking name whose de-qualified name matches the name you asked for. In this case, since you just cloned everything, you have every remote-tracking name with the `origin/` prefix, and thus there is exactly one such match, which is `origin/master`. The `checkout` command then *creates* this as a local branch, pointing to the *same* commit—the tip commit of the remote-tracking name—and as a bonus side-effect, marks this new local branch as *tracking* the remote-tracking name (the blue arrow in Figure 6.3).

[31] We will see later when and how Git chooses the name `master`.

Exercise 6.2: Obviously there could be none, e.g., if you ask to check out `numbat` when there is no branch for numbats yet. How could there be more than one?

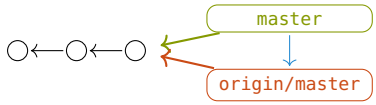

Figure 6.3: Git: create `master` via `origin/master`.

If you are perhaps a bit dizzy at this point with all the re-use of the words "tracking," "local," "remote," and "branch," rest assured that this is quite reasonable. You have a "local branch" that is said to be "tracking" a "remote-tracking branch"—or as I prefer to call it, a "remote-tracking name."[32] The local branch is not always called a tracking branch even though it is tracking something,[33] and the "remote-tracking name" is really a locally-stored and automatically-updated name, remembering where their `master` was the last time your Git talked with `origin`'s Git. Personally, I find this terminology awkward and cumbersome. It grew into this mess through historical usage, starting from very old versions of Git that lacked the entire concept of remotes. Nonetheless, it is the modern Git terminology and we must use it to talk with other Git users.

[32] If you use the phrase *remote-tracking branch name*, you will need the phrase *local branch name* to tell the two apart.

[33] This varies from one group to another. It seems quite reasonable to call these "tracking branches," and some people do, while others reserve the word "tracking" for remote-tracking names.

In summary, then:

- A *remote-tracking name* like `origin/master` is a locally-stored name that is qualified at the front with the name of a remote like `origin`. It is automatically updated on `fetch` and `push` (with some minor annoying details left for later) when your Git synchronizes with the remote `origin`. A remote-tracking name therefore remembers *where their branch-tip was the last time we talked with that remote.*

- A *local branch* like `master` is a locally-stored name that is *not* qual-

ified at the front. It may or may not *track* another branch name. If it does track another branch, it can track *either* a remote-tracking name *or* another local branch. We will show later what this Git-specific concept is about, i.e., what it means for one branch-name to *track* another branch-name.

- When asked to check out a branch by name, but no branch by that name exists yet, `git checkout` can create a *new* branch that points to the same tip commit as some remote-tracking name. It will do this whenever there is *exactly one* remote-tracking name that matches the name, such as `master` vs `origin/master`.

- The branch name that the initial `clone` checks out, usually `master`, is actually whatever the remote recommends. You can override this recommendation using `git clone --branch` *`branch`*, but (at least for now) there is never any need.

This is how Git manages to get you on a local `master` (or other branch) immediately upon cloning a repository. Note that *everything is local*, even if its name has the word "remote" in it. Only a few commands—chiefly `fetch` and `push`, and of course the original `clone`[34]—try to call up the remote. This is what *everything is local* means: you do all your work on your local machine, in your local repository, until you explicitly synchronize.

[34] This is not meant to be a full list, but most of the other commands you will use work entirely locally.

Mercurial: cloning existing projects

Mercurial's clone process is far simpler than Git's, since it simply clones the entire repository, automatically including all the branches, and then updates to the highest-numbered commit in branch `default`.[35] Note that the highest sequential number is automatically the most recent commit in both the original repository and your new clone, but *only because your clone is currently identical* to the original, and commits are added sequentially. Once you start adding your own commits, and the owner of the original repository continues adding commits there, your local revision numbers will stop matching up.

[35] As with Git, there is a way for the server to recommend a particular commit, maybe not even on `default`, and you can override this at `clone` time.

In other words, if you have just cloned Alice's repository and it has over 7400 commits, you can ask her about the Tasmanian devils in commit #7351, and she'll have the same `-r7351` that you do. However, a few months from now when you both have more than 7900 commits, your `-r7822` and her `-r7822` may be different. All the earlier commits are the same, so the initial clone can just go by number, but after a while your two repositories will have a different history of commits and only *some* local numbers will match up.

Cloning our three-commit repository

Let's see how all this works in practice by cloning the Git or Mercurial repository we made in the first part of this chapter. If you like, you can do this across a network using multiple machines, but this example will just use local files, so that we need not set up any servers.

We could start this way:

```
cd ../                      move up to dir containing project
mkdir project-copy          make a directory for the clone
cd project-copy             enter that directory
git clone ../project        or hg clone ../project
cd project                  enter the clone
```

The clone would made a sub-sub-directory named `project`, so we would wind up with `project-copy/project`, which seems a bit redundant. Instead, we can direct the `clone` sub-command to make the clone in a directory name of our choice:

Exercise 6.3: (Optional) Try doing this the long way here, just to see it in action.

```
cd ../
git clone project project-copy      or
hg clone project project-copy
cd project-copy
```

These print some reassuring messages and should then succeed. Git will tell you that it is done; Mercurial will tell you that it has done a `update`. Now let's use `branch` to inspect the clones.

Git: cloning our three-commit repository

For Git, we begin with:

```
git branch
```

The output is:

```
* sidebr
```

This is `sidebr`, not `master`!

We noted just a moment ago that the branch `clone` checks out (and therefore creates based on a remote branch) is suggested by the remote. So, let's see what *remote-tracking name* we have, using:

```
git branch -r
```

The `-r` option tells `branch` to show the remote-tracking names. We might expect to see two renamed items here, derived from the original `master` and `sidebr`, but in fact, we get *three* lines of output:

```
origin/HEAD -> origin/sidebr
origin/master
origin/sidebr
```

Here we have the two remote-tracking names we expected, but first we have this funny looking `origin/HEAD` and an arrow pointing to `origin/sidebr`. This shows us that the other repository's current branch is their `sidebr`. This is how they—we, really, since the origin repository is our own—are recommending that the clone use `sidebr` for its initial checkout: *The branch that is current in the origin repository determines which branch* `clone` *checks out.*

What this means in practice is that after cloning a repository, you should check which branches you and they have, and decide whether you want to be on whichever branch you are on now. This is even more important with older (pre-1.8.5) versions of Git, as they have to play a bit of a guessing game. Remember that `HEAD` is normally a symbolic reference, containing the name of another branch-name. That is, `HEAD` is a sort of arrow pointing to another branch name like `sidebr`. This is exactly what we just saw with `git branch -r`. Since Git version 1.8.5, the remote Git simply tells the cloning Git that `HEAD` points to `sidebr`. Before that, the remote Git told the cloning Git only that `HEAD` resolved to some specific hash. The cloning Git would look through all the incoming branches and pick one that had the same hash.

This is all of relatively minor importance, but if you understood all of the previous paragraph, you now know precisely how `HEAD` works. If not, try reading through this again, remembering that:

1. If `HEAD` is a symbolic reference, it contains some other branch name. Otherwise it contains a raw hash ID.

2. Any other branch name contains a raw hash ID.

3. Upon request, Git will turn any branch name into a hash ID.

4. Therefore, depending on the *kind* of request, Git can turn `HEAD` into *either* another branch name *or* a hash ID.

Git commands that want to know *What branch are we on?* get the *name* out of `HEAD`, while most commands that only want to know *What is the current commit GUID?* get the ID from `HEAD`, following through it to read the branch tip ID from the current branch as needed. The one very special command, `git checkout`, that can *put us on a branch or change branches* writes the new branch name into `HEAD`, and `git clone` ends by doing `git checkout` with the branch name from the remote.[36]

[36] Or, in Git version 1.8.4 and older, Git uses your `clone` command's best guess. This applies if *either* Git—local or remote—is older. Of course, if you supply a name with `-b`, `git clone` just uses that name.

Mercurial: cloning our three-commit repository

Unsurprisingly, Mercurial is once again far simpler than Git. We begin with `hg branch`, which simply prints:

```
default
```

The `hg branches` command is a bit more interesting:

```
sidebr                         2:4db4302bab15
default                        1:cd3c000e60f5
```

To really find out where we are, though, we need `hg summary`:

```
parent: 1:cd3c000e60f5
 add prototype kanga.c
branch: default
commit: (clean)
update: (current)
```

The only real evidence that we were just working in branch `sidebr` is that the *current* commit, sequence number `1` and hash `cd3c000e60f5`, is not the *highest numbered* commit.

Does this really matter? Perhaps not, but Mercurial's authors did eventually add a way to let the source repository tell the clone which revision to choose. If we go back to our original repository and create a bookmark named `@` pointing to its current commit—which, as with the Git repository, is the one at the tip of `sidebr` (in Mercurial, it's the *only* commit on that branch)—and redo the clone, we'll wind up on `sidebr`:

```
cd ..; rm -rf project-copy
cd project; hg bookmark -r . @
cd ..; hg clone project project-copy
```

This time the `clone` command says:[37]

```
updating to bookmark @ on branch sidebr
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

and sure enough, `cd project-copy; hg branch` now prints `sidebr`.

[37] Just as with Git, we can override the final `update` with a command-line argument, `-u` or `--updaterev`. We can even suppress it entirely with `--noupdate`.

Both: cloning our three-commit repository

If you are maintaining a Mercurial repository where new users should land on a different branch by default, it is up to you to set this `@` bookmark. As we just saw, the process can be automatic with Git since it uses the source repository's `HEAD`, which is also the source repository's current branch. In practice, it isn't automatic,[38] but we will cover this later.

[38] It's usually a good idea in Git to use a special *bare* clone as the URL target for push and fetch operations. Git users must manually set `HEAD` in a bare clone.

7
Working tree states: commits vs work-tree

In Chapter 3, we saw how using the checkout verb, which changes the *current commit*, replaced the contents of our work-tree with that from the specific commit we are checking out. In other words, if the current commit was `a234567` but is now `bcdef01`, the work-tree contents go from those for `a234567` (the then-current commit) to those for `bcdef01` (the now-current commit).

If we never did any *work* in the work-tree, this sort of behavior would be all there was to know. But we need to know precisely what happens when we *do* do something in the work-tree. We already did some work in Chapter 6, and we had to use `git add` more often in Git than we had to use `hg add` in Mercurial.

We also noted all the way back in Chapter 1 that there are files that we usually should not submit to the version control system. Some of these files will live in our work-trees.

By the end of this chapter, you will understand the role of the work-tree—and Git's index—in making new commits, and how to tell the VCS to ignore files it should not version. We will start by reviewing some facts about commits. Next, we will look at what happens as we modify the work tree, how Git's index stands in our way as we do this,[1] and how this all combines to eventually make a commit. Afterward, we'll look at what happens when we ask to check out a different commit *without* first committing pending changes.

[1] This index barrier has both positive and negative effects, as we will see.

Commits are forever . . . until removed

Commits are mostly-permanent, and definitely-unchanging. As we saw in Chapter 4, the hash ID of any commit is—must be—unique across all repositories that have the commit now, or ever will have the commit in the future. This means that once a commit is created, it can never be changed. It can, however, be *removed*, as long as nothing and no one else depends on it. That is, it must be a Mercurial-style head, with no descendant commits.[2]

[2] To remove an ancestor commit, we can first remove *all* of its descendants, so that it is this kind of head, then remove the commit itself. This is what Mercurial's `strip` command does.

We just saw that both Git and Mercurial provide an "amend" modifier we can use when committing. Amending a commit really means *discarding* the original commit, and switching to a new commit instead. In general, we only want to do this with *unpublished* commits. Mercurial enforces this for you automatically, though Git does not.

Suppose Alice publishes a commit to Bob, then "amends" the commit. In reality, she makes a new commit, and she stops using the original. This unwanted original goes away on its own: in Git, this happens eventually, and in Mercurial, it happens immediately. We will see the details about this process later. But the next time Alice brings commits back from Bob, she will probably bring her own old unwanted commit *back:* her VCS will think *Bob* supplied it as new-and-in-use! The precise details vary between the two VCSes, but the effect is the same.

In other words, commits can be removed—from the commit DAG and from the repository—if and only if no one else has and depends on them. If Alice really needs to remove the commit, but Bob also has it, Alice will have to get Bob to remove the commit from *his* repository as well. If Bob has built new commits atop Alice's, this makes a lot of work for him.

(It's more difficult for anyone—Alice or Bob, in this case—to "amend" a commit that is not the most recent one. Such a commit has descendants in the commit DAG, and those descendant commits depend on the the original. For instance, if there are three commits in a row that lead to the tip or head of her branch, and Alice wants to "amend" the third one back, she must *copy* the two descendant commits to new commits that now depend on the new "amended" commit. This same idea allows Bob to retain with his own work: if Alice *must* retract a published commit that Bob now depends upon, Bob can copy his existing commmits to a new set of commits, avoiding the commit Alice wishes to retract. It's best to avoid this entirely, but we will see some practical examples of it much later.)

In any case, we usually do not remove commits, but only add new ones. Mercurial used to be quite fussy about this, with Git being much more relaxed.[3] When we are only adding *new* commits, each existing commit is permanent, frozen forever in time. Both VCSes encourage us to think of commits this way, and to behave this way as well—except, that is, with just-made, definitely unpublished, amendable commits.

[3] This is in part because Git can secretly retain the commits, just putting them onto *no* branch. This allows you to restore or copy them later, as long as you do it before they expire. This course of action is not available to Mercurial.

Working trees are not permanent and can be clean or dirty

While *commits* cannot change, work-trees can and must. If we are to do anything new in our repository, we must work in our work-tree. This means that the files in the work-tree become *different from* those in the current commit.

If we have not changed anything in the work-tree, the VCS calls it *clean*. We can switch commits freely. Any files that need to be removed entirely can be safely removed, because they're safely saved in a commit. Any files whose contents must change can be safely changed, because those contents are safe in a commit. In other words, as long as the work-tree is clean, we can always switch commits.[4]

If we have modified some files, though, the work-tree is *dirty*. If we then direct our VCS to switch commits, what happens to our changed files? Git and Mercurial have somewhat different answers, but both systems try to carry the changes with them. For the moment, let's set that last idea aside.

[4] This is an overstatement, because we can have *untracked* and/or *ignored* files, as we will see.

The text below mentions sleepy koalas: they generally sleep for 18 to 22 hours a day. This is because the koala's primary food source—eucalyptus leaves—is both a poor source of nutrients, and rather toxic. Eating more food would obtain more calories for more activity, but poison the animal. The koala's solution to this dilemma is to expend as little energy as possible.

Mercurial: the work tree is like a changeable commit

We saw earlier that the first line of `hg summary` said that the *parent* was the current commit. Let's work with our side branch, in which we have created `koala.txt`. Here is what Mercurial has to say about it:

```
$ hg id
db6f6e1d8715 (sidebr) tip
$ hg summary
parent: 2:db6f6e1d8715 tip
 add prototype koala file
branch: sidebr
commit: (clean)
update: (current)
phases: 3 draft
```

Let's modify `koala.txt` to add a second line:

```
$ echo they also sleep a great deal >> koala.txt
$ hg id
db6f6e1d8715+ (sidebr) tip
$ hg summary
parent: 2:db6f6e1d8715 tip
 add prototype koala file
branch: sidebr
commit: 1 modified
update: (current)
phases: 3 draft
```

Note that the `hg id` output now shows a plus sign, marking the work-tree as dirty, and the `hg summary` output now says `1 modified`.

What's going on here is that Mercurial treats the work-tree as a *proposed commit*, a sort of as-yet-uncommitted commit. Since this proposed or prospective commit is not yet actually committed, we can change it all we want. But since it is *like* a commit, its *parent* is the actual current commit. This is *why* `hg summary` calls the current commit the "parent" of the work-tree state.

If we were to make a new commit right now, it would have one modified file mentioning sleepy koalas. If we modify the `README` file, `hg summary` will say `2 modified`. If we instead create a *new* file and fail to `hg add` it, `hg summary` will say `1 modified, 1 unknown`; and if we do `hg add` it, `hg summary` will say `1 modified, 1 added`. Note that at any point, `hg status` will show us the status of each of these (added, modified, or unknown) files.

It is worth mentioning here that if we are in the middle of a merge, `hg summary` will list *both* parents of the prospective merge commit. The current commit is always listed first. The second parent's ID is saved in a hidden data structure called the *dirstate*. We will come back to this later, when we cover the process of making merge commits and resolving merge conflicts, but the nice thing about this dirstate is that we can basically ignore it: Mercurial brings it up only when necessary.

Mercurial's `hg status` will list each modified file with a single uppercase M—modified—in front. There is only the one file, so this is not very interesting now, but let's go ahead and run it anyway. Then we should commit the sleepy koala:

```
$ hg status
M koala.txt
$ hg commit -m 'mark koalas lazy'
$ hg summary
parent: 3:ada3df2947f7 tip
 mark koalas lazy
branch: sidebr
commit: (clean)
update: (current)
phases: 4 draft
```

The current commit is now 3 and the work-tree is clean. Note that the work-tree is proposed commit 4, and we now have a clue regarding the last line of the summary: proposed commit 4 is in *draft phase*. This means it is not yet published—which is trivially true: the modifiable work-tree state *cannot* be published. There are now four commits in the repository, numbered zero through three. Number three is the *tip* commit, and this is what the word `tip` is doing at the end of the `parent` line. (Don't confuse Mercurial's tip—or highest numbered—commit with Git's tip commits, which are those pointed-to by a branch name.)

Git's index

We have mentioned several times that Git imposes an extra, intermediate, prominent yet semi-hidden data structure that Git calls the *index*, *cache*, or *staging area*. This index lives *between* the current or `HEAD` commit and the work-tree: see Table 7.1. You can copy files from the `HEAD` commit to the index, from the index to the work-tree, or from the work-tree to the index. Mercurial gets away without an index at all,[5] proving that the index is not necessary. Nonetheless, Git uses its index to help it keep track of changes made in the work-tree, and—as we will see in a moment—to help it ignore files that should not be versioned. Git also uses the index to address a third issue we will learn more about when we cover merging in greater detail. We will now see some of the things the index does for us that Mercurial cannot do. You can decide for yourself whether these features are worth the learning cost of Git's index. Unfortunately, even if you choose not to *use* these features, Git forces you to learn about them.

[5] Mercurial does have two data structures that it keeps much better hidden: the manifest, which as we already mentioned keeps a list of all files, and its dirstate, which we can largely ignore. Neither of these is quite like Git's index: in particular, they never get in our way.

While Git's index gets in our way all the time, it also provides several features, and is a key component of how and why Git is so fast, compared to similar VCSes.

		writable?	
file	`HEAD`	index	w.tree
`README`	✗	✓	✓
`koala.txt`	✗	✓	✓

Table 7.1: The index.

Git: the index contains the proposed commit

Let's make the same change in Git we did in Mercurial, observing the difference in `git status` output:

```
$ echo they also sleep a great deal >> koala.txt
$ git status --short
 M koala.txt
$ git add koala.txt
$ git status --short
M  koala.txt
```

This time, the letter `M` *moved* (and changed color, from red to green, if you have color enabled). If we used the longer form of `git status`, we would see that `koala.txt` went from `Changes not staged for commit` to `Changes to be committed`. This is because the `add` command *copies* the file, from the work-tree into the index. In Git, it's the *index*, not the work-tree, that is the proposed new commit.

This means, however, that when we work in the work-tree, *we are not working on the files that will be committed.* There is a hidden, *second set* of to-be-committed files. If there are two copies of each file, there must be *two* file statuses, and that is in fact the case. Without committing, let's change `koala.txt` *again*, so that it is different from both the current commit (which has one line) and the copy in the index (which has two lines):

```
$ echo third line, not to be committed >> koala.txt
$ git status --short
MM koala.txt
```

We now have *three different versions* of one single file `koala.txt` all active at the same time. The one line version is in the current commit, the two line version is in the index, and the three line version is in the work-tree. The `status` command runs *two* comparisons, and prints the two `M` letters. The first `M` says that the current, or `HEAD`, commit `koala.txt` differs from the index `koala.txt`. The second `M` says that the index `koala.txt` differs from the work-tree `koala.txt`.

Exercise 7.1: Use the longer form of `git status` now. Does it tell us that there are changes to be committed? Does it tell us that there are changes not staged for commit?

Let's do one more thing now before we commit: Let's make the *work-tree version* of the file match the *current commit* version. We can do this by putting back the original single line. Let's see what `git status --short` says about it, then make the commit:

```
$ echo koalas look cute and cuddly > koala.txt
$ git status --short
MM koala.txt
$ git commit -m 'mark koalas lazy'
[sidebr 0cbdbdb] mark koalas lazy
 1 file changed, 1 insertion(+)
$ git status --short
 M koala.txt
```

Now that we have made the commit, the *first* `M` is gone: the `HEAD` and index versions of `koala.txt` match. The second `M` remains: the index and work-tree versions of `koala.txt` do not match. The work-tree version *does* match the *previous* commit, but that does not matter to `git status`.

Git's index is not so easy to see, so use `git status`

You should ask: *How* do *we know what just got committed, or what* will be *committed?* The complete answer requires peeking ahead. For now, though, note that if the index and work-tree *match*, you can just look at the work-tree file. If they *don't*, you must either find a way to look at the index file, or *make* them match.

Exercise 7.2: How *do* we know what will be committed?

If we run the long form of `git status`, though, Git gives us some hints, right after it mentions `Changes not staged for commit`:

```
$ git status
On branch sidebr
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   koala.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

We already know that `git add` copies from the work-tree to the index. Using `git checkout -- koala.txt`, however, copies from the

index to the work-tree. In other words, it is almost the reverse of `add`.[6] Let's do that now, and then look at the status again:

```
$ git checkout -- koala.txt
$ git status
On branch sidebr
nothing to commit, working tree clean
```

[6] Note that this form of `checkout`, where we give it a path name, overwrites the work-tree file with the index version *without* first asking whether it should destroy our hard work. Be careful with this command!

Making the work-tree file match the index version made the work-tree clean. The index naturally matches the new `HEAD` commit we just made, and the only difference between the index and the work-tree was this one file, so now the index and work-tree match too.

Exercise 7.3: Try making the index version different from the `HEAD` version, by editing the work-tree version and using `git add`. Then, edit the work-tree version so that it matches the `HEAD` version. We already know that this will show up as short-status `MM`, or two separate changes in the long status. But now, use `git add` to copy the work-tree version—which matches the `HEAD` version—into the index again. What will happen to the status?

Summary of Git's index, so far

These, then, are the three things to know about Git's index at this point:

The index sits between `HEAD` and the work-tree. You must copy files from the work-tree into the index before committing. You can, however, also copy files from the index to the work-tree. Beware: when you request this, Git will happily overwrite your work-tree contents.

Each new commit is made from the index, not the work-tree. This is why you must `git add` so often. This does, however, let you commit a file version that is not in your work-tree. This may seem like a crazy thing to do, but we will see, later, how you can use this to *split* a change into several commits.[7]

Since you do have to do this so often, `git add` provides an en-masse `--all` option. Confusingly, this does not add *new* files unless they are listed on the command-line as well. The option essentially makes Git scan the *current* index contents: for each file that *is* in the index now, Git will copy the work-tree version into the index, updating what will be in the next commit. In Git version 2.0 and later, this will also notice any such files that you have *removed* from the work-tree, and remove them from the index. In other words, this option makes Git behave somewhat more like Mercurial—moreso in the older versions of Git, which do not remove files that have gone missing (we'll see in a moment that Mercurial calls such files "missing" and deliberately *does not* remove them).

[7] In Mercurial, splitting a single file change into several commits requires copying the file outside the VCS. Thus, this is one of the things Git can do that Mercurial cannot. Is this feature worth the complexity of the index? Maybe, or maybe not; but there are more.

There are a number of subtle traps here though, such as another difference in behavior in pre-2.0 Git when `--all` is used without path names while within a subdirectory of the top level of the repository.[8] I find it wise to use `--all` sparingly, and to be careful with `git status` afterward.

[8] For the curious, the effect of `--all` in Git before 2.0 was to scan only the current directory and subdirectories, and as already noted, to ignore what Mercurial calls "missing" files. In 2.0 and later, Git effectively scans the entire work-tree by scanning the entire index.

After making a commit, the index and `HEAD` *match.* Other parts of Git, and other documentation, sometimes refer to this state as being *empty*. This is wrong, or at least highly misleading. The index is not *empty;* it simply matches the current commit.[9]

[9] There are some tricky ways to run `git commit` that use a *temporary* index, so that in the end, the regular index and `HEAD` may not match. Still, the index is not *empty* at this point.

There is much more to learn about the index, but this will suffice for the moment.

New files, removed files, and untracked files

While Mercurial has no index, we can use Git's index to illustrate in more detail how new files get added and existing files removed, and what it means for a file to be *untracked*, in both VCSes. This is because, for once, Git's conceptualization is slightly simpler than Mercurial's (once we buy into the idea of an index, anyway). In addition, Git allows us to create and destroy entire branches with no penalty (as long as we don't publish them). This lets us test some examples, then hide all the evidence of that testing.

Suppose we make a new branch in Git, then add and commit a file named `removed`. We then run `git rm removed` to remove it from the index and work-tree. Next, we create a file named `new` and add it to the index, without creating a new commit. We also create a file named `utfile` without adding it anywhere, and remove the `README` file without telling Git that we did. Here are the shell commands to do this:

```
$ git checkout -b test
Switched to a new branch 'test'
$ echo remove me > removed
$ git add removed && git commit -m "add file to remove"
[test 0fd4240] add file to remove
 1 file changed, 1 insertion(+)
 create mode 100644 removed
$ git rm removed
rm 'removed'
$ echo new added file > new
$ git add new
$ echo untracked > utfile
$ rm README
$ git status --short
 D README
A  new
D  removed
?? utfile
```

The situation now is as shown in Table 7.2. The status output skips `koala.txt` since all three copies exist and are identical; but everything else has some change to show. Compare the `git status --short` output above to the table, and note that where there is a

	present?		
file	`HEAD`	index	w.tree
`README`	✓	✓	✗
`koala.txt`	✓	✓	✓
new	✗	✓	✓
removed	✓	✗	✗
`utfile`	✗	✗	✓

Table 7.2: Git file states in the index.

difference to show, it shows up in the *first* column if it is a change from `HEAD` to index, and in the *second* column if it is a change from index to work-tree. There are new letter codes as well: uppercase D for deleted and uppercase A for added. Mercurial uses a different letter—uppercase R for removed—but otherwise works much the same here. Meanwhile, the untracked file is perhaps considered doubly mysterious.

The files `README`, `koala.txt`, and `new` are all *tracked*, because they are in the index. The `README` file is missing from the work-tree, so it shows up as deleted from the work-tree (but not the index). The file named `utfile` is *untracked* because it is not in the index. Note that `removed` *is* in the current commit, and *is not* in the current work-tree, so `status` shows it as deleted in the index. An interesting thing happens if we put it back into the work-tree (but not the index) now:

```
$ echo back into work-tree > removed
$ git status --short
 D README
A  new
D  removed
?? removed
?? utfile
```

The file named `removed` is now both deleted *and* untracked, so it gets listed twice.[10]

Let's clean all this up now before we go on. We immediately encounter a problem:

```
$ git checkout sidebr
error: The following untracked working tree files would be removed by checkout:
        removed
Please move or remove them before you switch branches.
Aborting
```

Git is, this time, trying to be careful not to destroy any of our precious data. We'll come back to this case at the end of this chapter, but for now, there is nothing valuable here, so we can use the otherwise fairly dangerous `--hard` option with the `git reset` command:

```
$ git reset --hard
HEAD is now at 0fd4240 add file to remove
```

This tells Git to go back to the committed state, losing all uncomitted changes in both index and work-tree. The untracked file will remain untracked, but now we can switch branches, and we can just remove the untracked file manually whenever we like, and delete the `test` branch entirely:

Exercise 7.4: There are two states (✓, ✗) for each file in each of three places (`HEAD`, index, and work-tree). This means there are eight possible states in all, but Table 7.2 lists only five. What are the other three and what do they imply?

Exercise 7.5 (advanced): We already saw a status reading "MM" when a file was different between `HEAD` and index and then also different between index and work-tree. There are many more two-letter combinations. What are all of them, and what do they mean? (Consult the documentation. Some combinations occur only while in a conflicted merge, which we have not covered yet. The letter R, for renamed, is also for something we have not yet covered.)

[10] The Git authors elected not to use "`D?`" as a status, but that would be another way to show it here.

```
$ git status --short
?? utfile
$ rm utfile
$ git checkout sidebr
Switched to branch 'sidebr'
$ git branch -D test
Deleted branch test (was 0fd4240).
```

(the uppercase `-D` option does a forced delete, telling Git to discard the branch even if that will lose some commits).

Summary: what it means for a file to be tracked or untracked

In both Git and Mercurial, a file is *untracked* if and only if:

- it *is* in the current directory, but
- it *won't be* in the next commit.

In Git, it is the index that determines what is in the next commit, so it is the presence of a file in the index that determines whether the file is tracked. In Mercurial, it is the manifest that determines what is in the next commit.

As we already saw, Mercurial is like Git in one way: you must explicitly `hg add` a file for Mercurial to begin tracking it, by adding the file's path name to its manifest. Once a file is listed in the manifest, it *should be* in the work-tree, and *will be* in the next commit. When we removed the `README` file in Git, it did not affect the index, so that was harmless in one way, and confusing in another—the file does not show up in listings, yet it will be in commits. Fortunately `git status` tells us about the issue. If we try this same thing with Mercurial, its `status` will also complain:

```
$ rm README
$ hg status
! README
$ hg revert README
```

(the `revert` command in Mercurial gets the file back from the commit, similar to the slightly dangerous variant of `git checkout` when used with with a path name). This might seem a little odd—Mercurial's philosophy seems to be to use the work-tree as the next commit, so why doesn't it automatically remove files from the next commit if we remove them from the work-tree?[11] The answer is that it *did* do this at one time:

> You might wonder why Mercurial requires you to explicitly tell it that you are deleting a file. Early during the development of Mercurial, it let you delete a file however you pleased; Mercurial would notice the absence of the file automatically when you next ran a `hg commit`, and

[11] By the same token, of course, we could ask why Mercurial does not automatically add new files. Here, there is a better excuse: Mercurial can record a new file as a *copy of an existing file*, instead of merely "new". This affects the behavior of a later merge by changing the way that Mercurial identifies file-sets. We will leave the details for later, when we consider high- and low-level merge conflicts.

stop tracking the file. In practice, this made it too easy to accidentally remove a file without noticing.

[O'Sullivan, 2009]. Hence, if a file is listed in the manifest, but is not in the work-tree, Mercurial calls the file "missing".

Git's index vs Mercurial's state

Both Git's index and Mercurial's "the work tree is the next commit" ideas solve the question of read-only commit vs read/write work-tree. When using Git, you may modify the work-tree at any time without affecting what is or will be committed. You can then do an *en-masse* `git add --all`, which adds—and even removes, if necessary—everything you have changed.[12] You may then commit it all with a simple `git commit`. This almost makes Git as easy to use as Mercurial, where you simply modify the work-tree. Mercurial only needs a separate pre-commit command to add an entirely new file, or to remove a file entirely. Most of the time, `hg commit` alone suffices.

The index creates its own set of problems, though. In particular, its contents cannot be viewed easily,[13] and it mainly shows up in `status` output. It does, however, offer the ability to stage changes a little bit at a time. Mercurial's approach, of treating the work-tree as a modifiable proposed commit, is much more straightforward. You just edit and commit. What you see in your work-tree is what will be committed. If you want to commit something that is not in your work-tree, it's significantly harder: you must copy the files elsewhere, make the change in the work-tree, commit, and then copy the files back. The Git and Mercurial authors both consider their VCS's behavior here a point in their favor.

[12] Remember, the exact behavior of `--all` is somewhat different in Git version 2.0 and later than in earlier versions. In any case Git's behavior is further complicated by its index. Still, the general principle holds here.

[13] There are commands to inspect the index, and ways to look at the files in it, but this is nothing like simply looking at the work-tree.

Ignored files

With our purely text based Marsupial Maker, we have yet to come across a file that will live in our work-tree, but should never be committed. In real projects in most real languages, however, this occurs all the time. For instance, if we write Python code, Python compiles it to `*.pyc` files. If we write C or C++ code, the compilers generally write both `*.o` files and the final linked binary. Some editors also make editor temporaries and/or backup files in the same directory as the files being edited. OS X Finder creates `.DS_store` files in directories it shows as folders. As noted in Chapter 1, it's probably best not to commit any of these.

Both Git and Mercurial can be told about *ignored* files, which should *never* be committed. Here, Git once again gets much more complicated than Mercurial, and once again, the index is at fault.

As we saw, when either Git or Mercurial come across an unknown work-tree file, they complain about it. Their `status` commands print question marks, for instance. Both VCSes also offer a way to add *all* new files, or all files within a directory (including sub-directories). In Mercurial, you can simply run `hg add` or `hg addremove` with no arguments.[14] In Git, you run `git add --all`. To make these work correctly when you have files that should *never* be added, you must inform the VCS. The *add all files* commands will then add files *unless* they are *both* untracked *and* ignored.

[14] The difference between `hg add` and `hg addremove` is a bit subtle. Obviously, the latter removes files, just like `git add --all`. However, `hg addremove` also does *rename detection*, which we will describe later.

To do this, we list the files' names in *ignore files*. Git's ignore files are named `.gitignore`, and Mercurial's are named `.hgignore`. Listing the filename in one of these ignore files is necessary,[15] but it's not always *sufficient*. This is due to the notion of *untracked files* we covered in the previous section.

Exercise 7.6: Can an ignore file name itself?

[15] You can get away without it if you are willing to put up with constant complaints from your VCS and are very careful never to add these files by mistake.

Tracked files are never ignored

Almost everyone who uses either Git or Mercurial gets caught by this problem at some point: **If a file is tracked, it is not ignored.** Once a file gets into a commit, the file is automatically tracked. In Git, it's in the index (which initially matches each checked-out commit), and in Mercurial, it's in the manifest (which likewise initially matches each checked-out commit). Then, because the file is tracked, it will not be ignored, even if the VCS is told to ignore it. For some reason, this problem bites Git users much more often than it bites Mercurial users.

I think this is mainly because the index is so much harder to see than the work-tree, and partly because status information gets missed when large swaths of files are modified. Nobody really wants to look through a thousand lines of `status` output.

To get a file that is currently tracked to become both untracked and ignored, you *must* commit a removal of that file. This can cause other headaches later, so it is important to try to get this right initially. Later, we will see some techniques to mitigate the pain of improperly committed files. For now, though, just remember that *tracked files are never ignored.*

Which files are untracked?

Remember that for a file to be untracked, it must not be in the index (Git) or the manifest (Mercurial). How can we tell that this is the case? It's easiest to tell *before* we list the file's name in an ignore file.

With Mercurial, it's very straightforward: unless `hg status` says the file is missing, it's either tracked, or `hg status` gripes about it with a question mark. So if we don't see an exclamation point or a question mark, the file must be tracked.

With Git, it's not quite as straightforward, but `git status` may say one of three things:

- that the file is deleted in the work-tree (equivalent to Mercurial's "missing" status), in which case the file is tracked because it is in the index;
- that the file is deleted from the commit, in which case the file *was* tracked, but is no longer; or
- that the file is untracked, in which case we know it is untracked.

Hence, if `git status` says nothing, the file is tracked; if it says the file is untracked, the file is untracked; and if it says the file is deleted, we must be careful to see *where* it is deleted: if it is deleted from commit to index, it is becoming untracked, but if it is deleted from index to work-tree, it is still tracked. Alternatively, we can use `git ls-files --stage` to look directly into the index. This is a very useful diagnostic technique and you should remember it for hard cases. However, `git status` is much more useful for ordinary everyday work, and is what you should usually use.

Once an untracked file is listed in the appropriate ignore file, however, it becomes much more difficult to tell that the file is in fact untracked: both `git status` and `hg status` will say nothing if the file is *either* untracked *or* unmodified. If the file is untracked, it's being ignored, and if the file is tracked but unmodified, there is nothing to say.

Note that in both VCSes, any file's tracked-vs-untracked state can change as you move throughout the commit history. Just because a file is tracked or untracked *right now* does not mean it will continue to be tracked or untracked in the future, or will be tracked or untracked if we check out a past commit.

Making ignored files

Let's create some untracked files now. We did this earlier when we created `kanga.txt`, but this time, instead of using `add`, we'll put the file's name into an ignore file:

```
echo koala notes, not to be committed > koala.notes
echo koala.notes > .gitignore
```

This will ignore *only* the file `koala.notes` (though it will ignore *any* file whose *base name* matches, such as `subdir/koala.notes`). What if we want to ignore all `*.notes` files? The answer is that we can use just that:[16]

```
$ echo '*.notes' > .gitignore        overwrite
$ echo kangaroo notes > kanga.notes
$ git status -s                      -s = --short
?? .gitignore
```

[16] Remember to protect the asterisk from shell globbing, as we mentioned in Chapter 5, and are about to do here.

The `.gitignore` file is still untracked, but we should add and commit it:

```
$ git add .gitignore
$ git commit -m 'ignore *.notes'
[sidebr b6c0ebb] ignore *.notes
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Mercurial uses the same mechanism, but for once, it's more difficult than it is in Git:

```
$ echo koala notes, not to be committed > koala.notes
$ echo kangaroo notes > kanga.notes
$ echo syntax: glob > .hgignore
$ echo '*.notes' >> .hgignore
$ hg status
? .hgignore
```

As with Git, the ignore file is untracked; we should add and commit it:

```
$ hg add .hgignore
$ hg commit -m 'ignore *.notes'
```

There are not one but two lines in our `.hgignore` file. In particular, we had to start with a `syntax` directive. Mercurial defaults to using *regular expressions* (REs) in its ignore files. However, glob syntax is *far* easier to get right. For instance, if you have a `.hgignore` file containing only:

```
x*.o
bin/*
```

(forgetting the `syntax` line), Mercurial will, e.g., ignore files named `hello` and `binnacle`, as the first line literally means *any number of "x"s, followed by any character, followed by "o"* (`hello` has zero "x"s followed by "l" followed by "o"), and the second means *"bin" followed by zero or more slashes*. However, REs are more powerful than shell style globs.[17] In historic versions of Mercurial, REs used to work much faster as well.[18] If you understand REs and want to use them, go ahead, but I believe this is one of those rare cases where Mercurial's default (REs instead of globs) is wrong.

Git has no support at all for REs in its ignore files. Because REs *are* more powerful, this is a bit disappointing. There are some performance-oriented reasons for this, though. We will get into those details later.

[17] That is, all shell globs can be converted to REs, and in fact, this is what Mercurial does internally. On the other hand, there are REs that cannot be converted to shell globs.

[18] Most of these performance issues are fixed as of Mercurial version 3.1.

Renaming files

We will go into the subtler details of renaming in more depth later, but we must touch on renaming here, as it does affect work-tree

states. Internally, Git and Mercurial handle file renaming very differently. This is because, at a fundamental level, Mercurial *identifies* files by attaching an internal identifier (a unique number) when you first add the file to its manifest. This unique identifier is the file's *identity*, and it follows the file from then on. To rename a file in the work-tree, you must therefore run `hg rename` or `hg mv` or `hg move`.[19] Git offers a similar command, `git mv`, which seems to do the same thing, though as we will see later, the underlying implemention of file identity is quite different.

[19] These are all the same command, with multiple names to match both UNIX and Windows conventions. Use whichever one you like.

Let's do this now with our `README` file, on the side-branch:

```
$ ls
README          kanga.notes     koala.notes     koala.txt
$ git mv README README.md
$ git status -s
R  README -> README.md
$ git commit
    enter the commit message here, then write the file and exit
[sidebr 000d8ea] rename README
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename README => README.md (100%)
```

or in Mercurial:

```
$ ls
README          kanga.notes     koala.notes     koala.txt
$ hg mv README README.md
$ hg status
A README.md
R README
$ hg commit
    enter the commit message here, then write the file and exit
```

Git's status shows the file as code R for renamed, and includes both the old and new names. Note that the short-status R is in the *first* column: `git mv` renamed the file in both the index and in the work-tree, so the rename status is HEAD-vs-index (first column) and not index-vs-work-tree (second column.)[20] When we make the next commit, Git again shows the file as renamed, though this time with a mysterious percentage. We will see what this percentage means later (XXX when?).

[20] In fact, only the index copy of a file will ever show up as renamed, because when `git status` runs the *second* `git diff` to compare the index to the work-tree, it disables rename detection. You can, however, get an "RM" status, indicating that the index entry was renamed, and the work-tree copy under the new name differs from the index copy.

Mercurial's status shows `README` as *removed*,[21] and `README.md` as added, rather than showing a simple rename. It's not clear why—perhaps just because the Mercurial authors did not want to print two file names on one file-status line. In any case, Mercurial's `hg mv` renamed the file in both the manifest and the work-tree, retaining the file's internal ID number. Since there is no complicated index in Mercurial, there is only the one change to worry about. If you actually do this as separate remove and add steps, though, Mercurial

[21] Remember, Git uses D for deleted but Mercurial uses R for removed.

will assign the new file a new identity, and will not carry the rename operation through later merges. It's thus important that you use `hg mv` to do the rename (though if you forget, there is a way to recover, as long as you remember *before* you commit).

Changing the current commit without first committing

We know that all commits are, by definition, read-only. Their GUID hash IDs are determined by their contents (and as we saw in Chapter 4, all the commits *leading to* that commit as well): if you were to somehow change the contents, the commit would acquire a new, different GUID. We saw how this works with the `--amend` option: the commit gets pushed aside, replaced with the new one.

We have also seen that when we use the `checkout` verb to *change* the current commit, the work-tree contents are replaced with the contents of the target commit. This happens in both VCSes, both of which have a work-tree. This rule holds for Git's index too: checking out some other commit requires changing the contents of both the index and the work-tree. So the read/write areas—the work-tree, and the index if there is one—are clobbered, or at least partly overwritten, by a checkout operation.

If we do some work, then make a commit, this new read-only commit saves all the work we did, not just now, but forever.[22] But what if we have done a bunch of work, but have forgotten to commit it? What happens if we change commits *then?* Git and Mercurial have similar, but not quite identical, answers.

We already saw one example earlier in Git, when demonstrating removed-from-the-index (and thus untracked) files. Git tries hard to make sure we do not lose work here. But Git's behavior in other cases can be somewhat baffling. For instance, we are at the tip of our side branch `sidebr` right now. As compared to our `master` branch, we added a prototype koala file, marked koalas lazy, ignored `*.notes`, and renamed `README` to `README.md`. In our Mercurial repository, we did the same things (though the other branch is `default` rather than `master`). Let's add a new file, but not commit it, then attempt to switch branches:

```
$ echo new file > newfile.txt
$ git add newfile.txt
$ git checkout master
A       newfile.txt
Switched to branch 'master'
```

This is new: `checkout` seems to be running `status`. Well, almost—let's run our own and compare:

[22] This is certainly true in Mercurial, which has no index in the way. But what about Git, where the commit saves the index but not the work-tree? If the index and work-tree match, are you safe? What if they differ?

Except for untracked files, if the index and work-tree match, you are always safe after a commit in Git. The normal, user-facing commands also verify that nothing in the work-tree will be clobbered if it differs from the index.

In a fundamental sense, though, all file versions that are solely in the work-tree, whether this is simply *not yet* copied into the index, or untracked, or untracked-and-ignored, are less valuable to Git than those in the index, since they do not get committed, and hence have not become permanent residents of the repository.

Exercise 7.7: Predict output of the long form of `git status`.

```
$ git status -s
A  newfile.txt
?? kanga.notes
?? koala.notes
```

What happened? Why did Git let us change commits and branches without first committing our new file? Why is the new file now added in branch `master`? And where did these untracked files come from?

Remember that all of our changes are taking place in the index and the work-tree. None of these have been committed yet, and while the index is a *proposed* commit, it's not an actual commit. Git is able to change the current commit without having to touch `newfile.txt` in the index or the work-tree. So this is exactly what Git does: it moves your `HEAD` and replaces index contents where it must, but leaves the rest of the index alone. It replaces work-tree contents where it must, but leaves the rest of the work-tree alone. The output from `status` is not telling us that `newfile.txt` is added to a commit, but rather that `newfile.txt` is in the index and not in `HEAD`.

Similarly, the two notes files are untracked, i.e., in the work-tree but not in the index. Git did not have to touch them at all, so it didn't. But Git *did* have to remove the `.gitignore` file, which is in the tip commit of `sidebr` but is not in the current, tip of `master`, commit. So these files are now untracked but *not* ignored, and hence show up in the status.

If we *had* changed a file—in either the index *or* the work-tree—that Git would have had to replace during the checkout, Git would simply refuse the checkout, as we saw earlier. This means any changes that Git *does* carry across the checkout are usually trivial to carry back across a checkout back to the original branch. If you forgot to commit, you can return to your original branch, make your commit, and *then* switch to the branch you wanted to work on.

Exercise 7.8: Determine whether this kind of change-carrying checkout is *always* trivially reversible. Hint: enumerate the various states for each version of a file: modified in the index compared to `HEAD`, new in the index, modified in the work-tree, and so on. Remember that `HEAD` changes *twice* during the switch and switch-back process.

Let's clean this up by switching back to `sidebr` and then removing `newfile.txt` from both index and work-tree:

```
$ git checkout sidebr
A       newfile.txt
Switched to branch 'sidebr'
$ git rm newfile.txt
error: the following file has changes staged in the index:
    newfile.txt
(use -cached to keep the file, or -f to force removal)
$ git rm -f newfile.txt
rm 'newfile.txt'
```

As before, Git carries the extra index entry for the new file across the `checkout` step. It then tells us that a plain removal will lose data, i.e., that `newfile.txt` in the index differs from `newfile.txt` in `HEAD` (triv-

ially true since it doesn't *exist* in `HEAD`). The forceful removal works and the work-tree is clean. The two notes files are still untracked, but now Git reads `.gitignore` and knows not to complain about them, as they are untracked-and-ignored.

We can try the same in Mercurial, but it behaves differently:

```
$ echo new file > newfile.txt
$ hg add newfile.txt
$ hg checkout default
abort: uncommitted changes
(commit or update --clean to discard changes)
```

Since Mercurial lacks this intermediate index, it has no place to carry the uncommitted change. It simply rejects the attempt to change branches. We can remove the file from the manifest:

Exercise 7.9: Try using `hg rm newfile.txt` here instead.

```
$ hg forget newfile.txt
```

and it goes back to being untracked, after which we must remove it manually.[23]

[23] We could also use `hg update --clean default` to remove the added file and switch branches.

Just like Git, if we were to switch to the main branch and run the `status` command, the notes files would show up as untracked. But Mercurial has one other surprise for us. Let's modify `koala.txt` again, deliberately fail to commit it, and check out the commit from before we marked koalas lazy. This is revision 2 (we can find the number using `hg log`):

```
$ echo koalas have two thumbs on each front paw >> koala.txt
$ hg update -r 2
merging koala.txt
warning: conflicts while merging koala.txt! (edit, then use 'hg resolve --mark')
1 files updated, 0 files merged, 2 files removed, 1 files unresolved
use 'hg resolve' to retry unresolved file merges
```

What has happened here is that Mercurial has attempted to *merge* our uncommitted change to `koala.txt` into the version stored in revision 2.[24] The merge failed with a conflict. Since we have not yet covered the mechanics of merges, let's do a discarding update back to the tip commit:

[24] If you want it, Git *can* do this as well, using `git checkout --merge`. We will consider this more later.

```
$ hg checkout --clean -r tip
3 files updated, 0 files merged, 1 files removed, 0 files unresolved
$ hg status
? koala.txt.orig
```

This `.orig` file is left behind by the failed merge; we can just remove it now:

```
$ rm koala.txt.orig
```

The main takeaway here is to be careful in Mercurial: check your status before switching revisions, lest it attempt to merge existing changes with the target revision; This is also wise in Git, even though its safety checks and default checkout mode are a little safer than Mercurial's here.

A side note on configuration files

Projects that need configuration may come with a *sample* configuration, and/or a template configuration, and/or may create a real configuration during installation. But unless this configuration is strongly tied to specific *commits* (so that it should change on each checkout), it should be outside the versioned area entirely—as is true of Git and Mercurial configurations, for instance—or else should be left untracked-and-ignored.

(There are some borderline cases, and there are cases where Git itself gets this wrong. For instance, `.gitattributes` files are carried with each commit, which is generally the correct thing to do, but these can refer to *drivers* that are defined in uncommitted configuration files, which is wrong.)

Note that it is a good idea for complex configurations to allow for future extensions. There are many ways to handle this, including flexible text formats like those used by the VCSes, or version-numbered configurations with upgrade and downgrade operations. Note that any version numbers for the configurations are usually only *loosely* coupled to the versioning of the code that uses them.

Summary

The `status` command will show you the state of your as-yet-uncommitted *next* commit, as compared to your existing, permanent *current* commit. In Mercurial, this is simply your current commit "." versus your work-tree: there is nothing standing in the way between them. Since Git has its index in the way, this is your current commit `HEAD` versus your index, and Git will also show you the state of your work-tree as compared to your index: this is what you *could* copy into your index to change what will be in your next commit. Each file, indicated by a path name, can be in one of three states: *tracked*, *untracked*, or *untracked-and-ignored*. We normally shorten the last to just "ignored", but only untracked files can be ignored. Meanwhile, any file that is now tracked, or was tracked due to the current commit, can be newly added, modified, unmodified, deleted/removed, or renamed.

The work-tree as a whole—and in Git's case, the index that holds the next commit and that we generally change by copying *from* our work-tree—can itself be *clean* or *dirty*, based on whether any tracked files in it differ from their current-commit counterparts. If any files are new or deleted or renamed, the index or work-tree is automatically dirty until we commit even if the remaining files' contents themselves are unchanged. It's wise to check the status as carefully as you can before committing and moving on.

Mercurial will attempt to merge uncommitted work-tree changes when switching commits, so be sure to check your status, and do not commit files that should be ignored.

If you commit a file that *should have* been ignored, that file is now tracked, and will be tracked every time that commit is extracted via `checkout`. If you move from a commit where the file is tracked to one where it is not, the VCS will *remove* the file.[25] If you do this by mistake *and have not published the commit*, you can fix it. You already know how to "amend"—really, replace—the most recent commit. We will see some techniques for more extensive history editing soon (XXX when?), and cover ways to take such files out of commits without losing them entirely (though usually the easiest way is just "copy somewhere else").

[25] As always, this means removal from the work-tree in both Git and Mercurial. Git, of course, removes it from the index as well when you move from a commit that has the file to one that does not.

8
Merges

In Chapter 3, we took a high level look at merging. Specifically, we saw that the point of a typical merge is to combine some series of changes from two or more lines of development. We can now look at this process in detail, including things that can go wrong and what there is to do about them. There are some fairly major differences between the underlying methods used in Git and Mercurial, too.

By the end of this chapter, you will know how to tell Git and Mercurial to make merges. You will know what a *merge conflict* within a file looks like, and what to do about it. You will see an example of a *high level* merge conflict, how these differ in Git and Mercurial, and what to do about them. Because Git leaves many implementation details somewhat exposed, you will see that the way Git records an ongoing merge and its conflicts. (Mercurial keeps these well hidden, making it much easier to use, though less flexible in some uncommon cases.) We will also revisit Git's fast-forward and squash cases.

An easy merge, in Git and in Mercurial

Before we dive into the mechanics of merging, let's use our existing repository-so-far to merge our koala work into our main-line branch, `master` or `default`. In Git, we need two commands:

```
$ git checkout master
Switched to branch 'master'
$ git merge sidebr
Removing README
```

Git's remark about removing `README`, which is shown here, may become invisible, wiped out by the editor. Depending on your system, editor, and other configuration items, it may re-appear once you exit the editor. It's not important either way, since Git repeats the information in the merge summary.

This brings up our editor on the initial message:

```
Merge branch 'sidebr'

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
```

```
# the commit.
```

We need not actually enter anything here, if this default message is sufficient.[1] We can just exit the editor without re-writing the file at all, after which Git produces this merge summary:[2]

[1] It's not. In fact, this message is especially bad, and we will fix it soon.

[2] Note that Git now repeats the message about removing `README`, which is why losing one earlier is not that important.

```
Merge made by the 'recursive' strategy.
 .gitignore | 1 +
 README     | 1 -
 README.md  | 2 ++
 koala.txt  | 2 ++
 4 files changed, 5 insertions(+), 1 deletion(-)
 create mode 100644 .gitignore
 delete mode 100644 README
 create mode 100644 README.md
 create mode 100644 koala.txt
```

Note that Git has already made the final merge commit.[3] When we try this same sequence in Mercurial, it will just print a reminder that we *should* commit, perhaps after carefully inspecting the merge result. We mentioned this difference in Chapter 3, and the likely reasons for it: Mercurial's `--amend` came relatively later in its development, so it still pauses in between if you discover you need to make some changes that the VCS did not itself make automatically.

[3] If we *do* discover that the merge result is incorrect, we can fix it in a subsequent, ordinary, non-merge commit. Or, we can fix it and amend the merge. Or, using Git's `--no-commit` option to make Git behave like Mercurial, we can *delay* the merge commit until after we have checked the result and made any necessary fixes. There's no one correct way to deal with this. We will see more about this later.

Let's do the same merge in Mercurial now:

```
$ hg checkout default
2 files updated, 0 files merged, 3 files removed, 0 files unresolved
$ hg merge sidebr
3 files updated, 0 files merged, 1 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

We can commit now in Mercurial, to make it behave more like Git; let's do that for the moment. As with Git, this brings up your chosen editor, giving you a chance to write a merge message. Unlike Git, however, the initial merge message is *empty*, and we must write one:

```
$ hg commit


HG: Enter commit message.  Lines beginning with 'HG:' are removed.
HG: Leave message empty to abort commit.
HG: -
HG: user: Chris Torek <chris.torek@example.com>
HG: branch merge
HG: branch 'default'
HG: changed .hgignore
HG: changed README.md
HG: changed koala.txt
HG: removed README
```

Let's use `merge initial koala work` as our message, then write out the file and exit the editor. Mercurial makes the commit, and we now

have the same things—the same contents for our two branches—in either VCS. In other words, despite some minor differences such as two vs three commands, the *merge result* is the same in both systems. Both find the same merge base commit; both combine the same sets of changes; and both commit the same result. (Mercurial did force us to write a much better merge *message*.)

What's in a merge commit

Our new commit has the same standard metadata as any non-merge commit: an author name, email, and date; the same for a committer, in Git but not Mercurial; parent commit IDs (two IDs this time, instead of just the usual one); associated source tree; and a log message. Let's look at how these VCSes show the merge:

```
$ git log
commit 3d8e089219d8a813b3907a511a9e31b70adc0f7e
Merge: 3c345b0 000d8ea
Author: Chris Torek <chris.torek@example.com>
Date:   Sat Aug 19 17:19:17 2017 -0700

    Merge branch 'sidebr'
```

Git's output is fairly terse, but does note that this new commit is a merge, using this extra `Merge:` line. Git prints an abbreviated hash for each of the two parents, and otherwise shows us the commit as usual.

```
$ hg log
changeset:   6:077bc776d123
tag:         tip
parent:      1:d05b1df8b8f6
parent:      5:5f5df3fc4f1c
user:        Chris Torek <chris.torek@example.com>
date:        Sat Aug 19 17:40:48 2017 -0700
summary:     merge iniital koala work
```

As with Git, Mercurial's output strongly resembles that for a non-merge commit, except that it prints both parent revision numbers (as both locally-sequential number and abbreviated hash).

We can see what's actually in the merge by looking at the work-tree. In Mercurial, the work-tree *is* the proposed next commit, which should exactly match the current commit, so this is automatically true. In Git, the *index*, not the work-tree, is the proposed next commit, but a successful merge updates the work-tree so that all of the tracked files in the work-tree match their versions in the index. Of course, if you're not quite sure, you should use the VCS's `status` command to double check: if the status is clean, the work-tree matches the index and the index matches the `HEAD` commit (Git), or

the work-tree matches the current commit (Mercurial), and hence what's in the work-tree is what's in the commit.

Most of the changes we brought in from the side branch were simply creating new files. This is one of the easiest cases for a merge: the VCS simply notes that the file did not exist in the merge base, still does not exist in one of the two branch tips or heads, and does exist in the other branch tip or head. The correct result of such a merge is to take the new file, and that's what Git and Mercurial did here.

Note that both Git and Mercurial have claimed to have deleted/removed `README` and created or modified `README.md`. Here, Git says it *created* the file, while Mercurial says it *modified* the file. This is because Mercurial managed to *identify* the file named `README` in the main-line branch with the file named `README.md` in the koala branch. At a high level, Mercurial saw that one of the changes we made on the side branch, to *rename* the file, should be brought into the main line.

This idea of file identity, which we noted all the way back in Chapter 1, is one of the keys to proper merging. Recall the brief discussion of renaming work-tree files from Chapter 7, page 130. In Mercurial, the rename is something we recorded when we ran `hg mv`. Although we used `git mv` to rename the file in Git, Git did not actually *record* a rename at that time. Instead, it attempts to *detect* the rename *now*, when we do the merge. In fact, Git *failed* to detect the rename, so it really *did* delete `README` and create `README.md`.

Fortunately, the effect was the same. Most of the time, these mechanics don't matter that much, but we'll soon use this same technique, where Git gets this wrong while Mercurial gets it right, to show one of the ways merges can go wrong. For now, just note that file create, delete, or rename operations occur at a sort of higher level: they affect the *set of files* in the commit, rather than the *contents* of any one particular file. It's in these high level operations where Git and Mercurial differ the most.

First and second parent

A DAG makes no special distinction between multiple parents of a commit node, but Git and Mercurial do: the *first parent* is always the commit that was current when the merge occurred. In other words, following just first parents follows the main line of the branch's development.

Mercurial does not need this property as often as Git, because each commit records its branch name: the merge we just created on branch `default` has one parent that's also on `default`—this is the one numbered `1:d05b1df8b8f6`—and one parent that's on `sidebr`.

Looking at the graph later, it is obvious that we merged `sidebr` into `default`.

In Git's case, the side-branch name appears only in the commit message, which of course we could have edited to get rid of the word `sidebr`. Moreover, now that we have merged the koala work, we can *delete* the branch name `sidebr`. As we saw in Chapter 2, page 41, our koala commits are now on *both* branches. The name `sidebr` is a spectacularly bad branch name. Deleting it leaves all our commits intact, retained through the `master` branch. That might be a good idea. We can also amend our unpublished merge commit to improve the commit message. Let's do that now, while we are sure it's unpublished, using the better commit message we wrote for Mercurial:

```
$ git commit --amend -m 'merge initial koala work'
[master edb2c66] merge initial koala work
 Date: ...
```

We can now delete the bad branch name:

```
$ git branch -d sidebr
Deleted branch sidebr (was 000d8ea).
```

In Mercurial, we are stuck with the bad branch name, but now that we have deleted it in Git, how do we know which commits were made on the "main line" `master` branch, and which were brought in via merging? This is where first vs non-first[4] parent lineage comes in.

[4] Git's merge commits allow more than two parents. We could say "first vs second" here, but the general case in Git is first, or not-first.

Let's take a look at two different *graphical* `git log` outputs, using the `--oneline` option:

```
$ git log --graph --oneline
*   edb2c66 merge initial koala work
|\
| * 000d8ea rename README
| * b6c0ebb ignore *.notes
| * 0cbdbdb mark koalas lazy
| * 49d5fae add prototype koala file
* | 3c345b0 add a kangaroo
|/
* 5318e61 initial commit
$ git log --first-parent --graph --oneline
* edb2c66 merge initial koala work
* 3c345b0 add a kangaroo
* 5318e61 initial commit
```

By directing Git to ignore the non-first commits, we get a shortened history, showing only the "main line" commits, including the merges that brought in other work. In other words, we see *only the merge itself.* Even if the work we did on the side or feature branch was very complicated, it simply appears on the main line in its final form.

Exercise 8.1: What if the work is incomplete? Is it a good idea to merge it yet? Think of all the reasons you can to merge earlier or later.

Even in Mercurial, though, both parents may come from a single branch. That is, we can create several heads—Git-style branches—

within a Mercurial named branch, and then merge two of them. We will see several cases of this soon. In this case, you can run into the same the problem of deciding, later, which parent was which at the time of the merge. Since Mercurial does exactly the same thing as Git, the follow-first-parent method works for both systems. However, for whatever reason, Mercurial's function that first-parent following is well-hidden, using the underscore-prefixed keyword `_firstancestors`. This may be because users are mostly assumed to use branch names, rather than tricky first-parent notions. Except for their permanence, branch names are certainly the more friendly way to work here.

Exercise 8.2: You might try to use Mercurial's repository-level numbering to guess which parent is which. When does this work, and when does it fail?

BEGIN OLDER STUFF

Many-parent merges

A non-merge commit normally has exactly one parent.[5] Mercurial limits commits to *at most* two parents, and any two-parent commit is a merge. Git, on the other hand, allows a commit to have two *or more* parents. Git calls these three-or-more-parent merges *octopus merges*. Of course, even in this kind of merge there must still be a second parent, and that suffices to mark the commit as a merge. Moreover, there is nothing really special about an octopus merge: While Git allows you to merge both `B` and `C` into `A` in one step, Mercurial allows you to merge `B` into `A`, and then merge `C` into `A`. These two pairwise merges will produce the same end result when there are no conflicts. When there *are* conflicts, Git will refuse to do an octopus merge anyway.

[5] Remember, though, that a root commit has *no* parents.

We will do our two merges below as two separate steps, but in Git, we could try merging Alice's wombats and Bob's kangaroos into the `develop` line in one octopus merge. In theory, this might help emphasize the unprivileged status of the two feature branches. In practice, octopus merges seem mainly useful for showing off your Git-fu. And, in any case, we have arranged for Alice and Bob to have a merge conflict.

Finding merge bases

Recall the definition of Lowest Common Ancestor from Chapter 2. We mentioned then that the LCA is the *merge base*. Both Git and Mercurial compute the merge base in the same way, by finding an LCA node in the commit DAG, using the current and other commits as the starting point for this search. For an octopus merge, Git uses all the input commits. That is, we simply generalize the two-node LCA algorithm to many nodes, finding whichever ancestors are common to *all* of them, and then use the lowest.

We also mentioned that in some complex DAGs, there may be *more than one* merge base. Git and Mercurial handle this case differently. We will address Git's method soon. Mercurial takes a very simple approach here: it simply picks one LCA node at random. In most cases this works well enough, and multiple LCAs themselves are rare enough, that this simple approach is usually fine. It is also much easier to describe, so we will assume for the moment that there is a single LCA to use.

figure for sequences of merges with merge bases goes here

Figure 8.1: Sequence of merges, with stitch pattern.

Before we go on to look at the mechanics of merging, consider Figure 8.1, in which several topic branches are *repeatedly* merged into a branch where we aggregate work for the next release. For instance, we might have Alice (or a whole group) working on wombats and Bob (or another group) on kangaroos, and we merge their work back to the overall development branch at regular intervals, or whenever it is deemed ready for internal testing, or whatever other criteria we choose. Note that each time we do a *new* merge, the merge base for that merge already includes all the work leading up to the point of the *previous* merge from that branch.

This is a significant factor in determining both when and why we merge. The more often we merge, the smaller the divergence at the *next* merge. Of course, each merge introduces some changes, which may disrupt other people's work—which itself is a reason to have them work on topic branches, as that keeps them isolated from the merge until *they* are ready to pick up others' work. Also, each merge takes time and effort that could go towards fixing bugs or developing new features. All of these play a role in determining how often to merge, but the fact that a *subsequent* merge will have a new merge base is the principle reason to make multiple merges.

figure for criss-cross merge goes here

Figure 8.2: Criss-cross merge.

Later, we will see tools that let you defer or avoid merging, and consider cases where these are better plans (if, e.g., the short-term gains from avoiding the merge outweigh any long-term gains from merging). We will also revisit this idea several times when we consider such items as over-eager merges,[6] bad merges, and *criss-cross merges*. Criss-cross merges occur when you merge a topic branch into a next-release branch, but then also merge the next-release branch back into the topic branch (see Figure 8.2). These criss-cross merges can produce DAGs with multiple LCAs. There is nothing fundamentally wrong with this, but you will need to know how it affects future

[6] "Over-eager" is not a technical term. I use this adjective to describe a merge made too early, before all the work in, e.g., a feature branch is actually ready, so that some or all of it must be backed out.

merges.

Finding changes since the common base

Having found the merge base, the VCS then computes two changesets: one from the merge base to the tip of the current commit, and one from the merge base to the tip of the other commit. In Mercurial's case, it already stores changesets, so all it has to do is aggregate the changes that lead from the base to the two commits in question. In Git's case, it stores snapshots, so it must now—on demand—produce two diffs, comparing the base to each commit. Here we also run into another difference between the two VCSes: Git stores only *file contents*, and must *guess* at any renames that may have occurred, while Mercurial records changes to directories and knows for certain whether `dir/file` was renamed, or even deleted and re-created. Git's guesses are based on its *similarity index*, whose computation is a bit complex and can be adjusted with various flags if needed.

These two changesets drive all of the automated merge action, so it is important for the VCS to get them right. Both systems have advantages: Git can find a rename even if the user failed to record it properly, and handles files that were improperly deleted and then resurrected, while Mercurial finds renames when Git fails to do so. Which method is better depends on your particular usage. In the end, both seem to be about equally effective.

Combining changesets

The point of getting the two separate changesets is to allow the VCS to combine them. Our goal—or at least, what the VCS assumes is our goal—is to keep *one copy* of each change introduced into some file.

For instance, suppose Bob is running a merge to bring in Alice's changes. Suppose further that Alice fixed a bug in `wombat.c`, but that both Alice and Bob noticed recently that some other file (such as `doc.txt`) contained the misspelling "woombat". Both removed the extra "o", so same file is changed between their common merge base and both Alice's and Bob's more-recent commits.

Both Git and Mercurial generally operate line-by-line when using these comparisons. They therefore show this change as:

```
 the ability of
-the woombat to move at high speed,
+the wombat to move at high speed,
 so that
```

(though both VCSes keep several additional lines of context). Since

both Alice and Bob made the *same* change to the *same area* of the *same file*, both VCSes will keep a single copy of this change.

Alice's fixes to `wombat.c`, on the other hand, have no counterpart in Bob's changes since the common merge base commit. Both VCSes will use the context of the base-to-Alice diff to find where Alice's changes should go into `wombat.c` (in case Bob has made other changes that have moved the lines around).

Doing a simple, unconflicted merge

We will play the part of Carol[7] who is tasked with combining Alice's and Bob's branches into the `develop` branch. Furthermore, we have set all this up so that Alice, Bob, and Carol all started with the *same* commit graph.[8] Both Alice and Bob have added new commits, but on different branches. We have also arranged to have a conflict when we merge Bob's work, but we will do our first merge (of Alice's work) without one.

As Carol, we start by obtaining (fetching in Git, pulling in Mercurial) both other branches, and checking out the `develop` branch. The exact commands will depend on the VCS—and in Git, whether everyone shares work through a central point—but the approach is the same.

For Git, if we are working without a central server, we would use:

```
git fetch alice                    get Alice's changes
git fetch bob                      get Bob's changes
git checkout develop               get ready to merge
git merge --no-ff alice/wombat     bring in Alice's changes
git merge --no-ff bob/kangaroo     attempt to bring in Bob's
```

(we'll see what `--no-ff` is about soon). Remember that Git renames branches, so Alice and Bob may work directly in `wombat` and `kangaroo`, but Carol must now refer to their work as `alice/wombat` and `bob/kangaroo`.

If Alice and Bob push their work to a central server named `origin`, Carol might use this instead:

```
git fetch origin                   get everything
git checkout develop               get ready to merge
git merge --no-ff origin/wombat    bring in Alice's changes
git merge --no-ff origin/kangaroo  attempt to bring in Bob's
```

For Mercurial, we might use:

```
hg pull alice
hg pull bob
    or
hg pull default      or whatever name Carol uses for the server
```

then, after all `pull` *commands:*

[7] This is actually a somewhat unusual method. More typically, whoever goes second—either Alice or Bob, but not both—would have to merge or rebase just her or his work. We'll see this in a moment.

[8] This is so that we can predict what will happen as we go along. If they all started with different commit graphs, we would need extra steps to resynchronize, and along the way, we would likely find different merge bases, hit different conflicts, and so on.

```
hg update develop
hg merge wombat
hg commit -m 'merge branch wombat into develop'
hg merge kangaroo
hg commit -m 'merge branch kangaroo into develop'
```

Since Mercurial does not rename branches, it does not matter whether we pull the two branches directly from Alice and Bob, or from a shared server.

When using Git, Carol might prefer to *fast-forward* her own local `wombat` and `kangaroo` branches, so that she does not have to type `origin/`. This makes the command sequence longer, though, and the only real difference it makes is to change Git's default merge-commit log message: "merge branch wombat into develop" vs "merge remote-tracking branch origin/wombat into develop". We will see more about fast-forward soon, and—unrelated to fast-forward—some short cuts to avoid typing names like `origin` or `alice` or `default` as often. Meanwhile, for now, let's assume that we can use just the names `wombat` and `kangaroo` for the two branches (or you can mentally replace each name with an appropriately-prefixed name).

Again, we expect a conflict on the second merge, when we try to bring in Bob's changes. For now let's focus on what happens with Alice's. Note that regardless of which VCS we are using, there are really just three steps to the the first merge: Obtain the commits to be merged; check out the branch that will hold the merge; and perform-and-commit the merge.

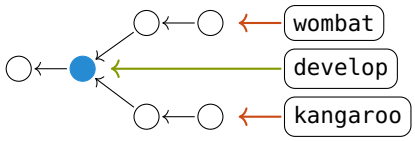

Figure 8.3: Commit graph before first merge

Carol's current commit graph, or at least an interesting portion of it, is shown here in Figure 8.3. The commit drawn in blue will be the merge base when Carol runs `merge wombat`. (Note that in Git, both Alice's and Bob's branches descend from this base commit, and it is *on all three branches*. In Mercurial, no commits share branches. The head of `develop` is still the branch-point for the other two branches, but unlike for Git, this has no particular consequences.) When Carol instructs the VCS to do the first merge, the VCS finds the LCA of the current commit (the Git-tip or Mercurial-head of `develop`) and the other commit (the tip or head of `wombat`), which is that blue commit. The VCS therefore does not even need to diff the merge base against the current commit—these are the same commit so this diff is trivially empty—nor does it really need to diff the merge base against

the other commit.[9] The combined diffs will just be the second set of diffs, and the result of applying the combined diffs to the current commit will be identical to the work tree associated with the tip (Git) or head (Mercurial) commit of `wombat`.

This triviality is in fact why we must supply the `--no-ff` option to Git. Specifically, when the merge base *is* the current commit, Git will normally not do a merge at all,[10] and instead *fast-forward* the current branch label. In this particular case, the result would be to change the `develop` branch label to point directly to the commit at the tip of `wombat`. The Git documentation calls this a "fast-forward merge," but this is something of a misnomer, since a fast-forward is not a merge at all.[11] When you want to force a merge, you should use the `--no-ff` option. It does not hurt to supply it every time you want a real merge, since it is does nothing if Git was already going to do a real merge.[12]For some workflows—including the one illustrated here—you may want to force merge commits when bringing specific topic or feature branches into a release or mastering branch. In our case we also want to force a real merge simply to illustrate the merge!

There is no corresponding flag in Mercurial since its commits are permanently affixed to a single branch. The notion of moving a branch label is simply nonsensical in Mercurial, and there is no such thing as a fast-forward.

[9] Mercurial must still build up the combined diffs, though, since the final changeset for the merge will be whatever it takes to convert from the blue commit to `wombat`.

[10] This behavior is suppressed not only with the `--no-ff` option, but also by default when the other commit is named via an annotated tag. Annotated tags have no counterpart in Mercurial and are left to Chapter XfutureX.

[11] Fast-forward operations are not just useful, but in fact crucial, for certain cases in Git. We will see these later.

[12] If you want to *suppress* a merge, Git offers `--ff-only` as well. Git's default action is to fast-forward the label if possible, and do a merge if not, so `--ff-only` is actually just a safety net: it makes the merge command fail if fast-forwarding is impossible.

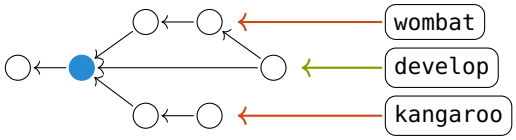

Figure 8.4: Commit graph after first merge

In any case, once Carol has merged Alice's work, her new commit graph is the one shown here in Figure 8.4. Git makes the new commit automatically, running Carol's preferred editor so that she can edit the log message. Mercurial makes Carol run `hg commit`. The new commit has two parents: the previous tip (Git) or head (Mercurial) of `develop` and the merged-in other commit at the tip/head of `wombat`. The *source tree* associated with this commit matches that in Alice's final commit. Git's automatic log message is, or is similar to, "merge branch wombat into develop" or "merge remote-tracking branch origin/wombat into develop" (this is somewhat Git-version-dependent, and configurable as well).

This particular merge will always succeed for the same reason that Git will try to do a fast-forward instead of a merge: Alice's `wombat` work starts from the tip/head of `develop`, so by definition there are no conflicting changes on the "ours" side of the merge.

Exercise 8.3: We claimed earlier that merges do not just take one side or the other of the merge, but this merge *did* just take Alice's side. Why?

Conflicts and conflict resolution

Now Carol will ask the VCS to merge Bob's work. The command is exactly the same except for the branch name, but this time, we had Alice and Bob make sure that we get a conflict, so that we can resolve it.

The diffs *for a single file* are like a set of instructions for doing delta compression: delete something here and/or insert something else there. If the change is purely deletion or purely insertion, we have just that one directive; if the change is a replacement, we have a deletion immediately followed by an insertion.[13]

When the VCS finds, in the two sets of deltas for some file, the *same* deletion-and/or-insertion sequence (disregarding exact position whenever necessary, but always accounting for context), this is a change that appears in both lines of development and the VCS takes just one copy. When the VCS finds different deltas that *do not* conflict, the VCS takes one copy of each. If, however, the VCS finds *different* deltas that apply to the *same* lines (after accounting for context), it declares a conflict. This can also occur even when the delta is the same, if the two contexts differ. For instance, this "same change but different context" conflict occurs when one side has the change just before the end of a file, and the other side has the change not so close to the end (because the second version of the file is longer).

This kind of conflict—the both-sides-modified, changes-collided case—is called a modify/modify conflict. These are the most common conflicts. Besides this case, there are two more cases that may cause the VCS to declare a conflict and stop. These are:

1. Create/create (or add/add) conflict. If a file did not exist in the base commit, but does exist in both the current and other commits, the VCS does not know which version of the file to use. (If both new versions match exactly, Git will simply take one of them. Must test this case in Mercurial.)

2. Modify/delete or rename/delete conflict. If the file did exist in the base, but the file was modified (and/or renamed) on one side and deleted on the other, the VCS does not know whether to keep the change, or delete the file.

Note that it is not possible to have a create/modify conflict, as the first implies that the file did not exist (was created on one side) and the second implies that the file *did* exist (was modified on the other side). For the same reason, create/delete is impossible as well.

The mechanisms Git and Mercurial use to record which files have conflicts are different, but the kinds of conflicts recorded are the same, and both VCSes will stop with the merge partly done and

[13] The actual deltas, if any—remember that while Mercurial uses changesets internally, Git only stores deltas in pack files—need not be line-oriented like this. It is only the merge process that, by default, works on a line-by-line basis.

make you fix up the mess. At this point, you must use something—it may be as simple as your file editor—to resolve each conflicted file, and then tell the VCS that the file is resolved.

Resolving conflicts manually

Let's consider a simple case of a merge conflict, where Alice and Bob both fixed one line in a file, but made two different fixes.

As before, the merge base is the blue commit (which is the same merge base as before), but this time, the set of changes from the merge base to the current commit are *not* empty. In fact, these are the same changes that Carol just brought in from Alice. The VCS also finds the diff from the merge base to the tip (Git) or head (Mercurial) of `kangaroo`, and attempts to combine them. Whenever Bob changed a file that Alice did not, or their changes do not conflict, the VCS combines them successfully, but for file `doc.txt`, we find that Alice fixed some incorrect documentation, but Bob changed both code *and* its documentation. Carol opens `doc.txt` in her editor and sees the VCS's result:

```
Some stuff here
that is the leading context.
<<<<<<< HEAD
Only red kangaroos are supported.
=======
Both red and gray kangaroos are supported.
>>>>>>> kangaroo
More stuff goes here
that is the trailing context.
```

The `<<<<<<<`, `=======`, and `>>>>>>>` markers are called *conflict markers*, and they surround the conflicting text. There is one thing missing here: what was in the base file before Alice and Bob changed it? Both Git and Mercurial can show you the line that was in the merge base version of the file, and I recommend enabling the option that does this.[14] To enable this in Git:

```
git config --global merge.conflictStyle diff3
```

To enable it in Mercurial, run `hg config --edit`, then use your editor to set `merge` to :merge3 in the `[ui]` section:

```
[ui]
merge = :merge3
```

Once you have this set, the merge will show this instead:

[14] This option can produce suprising results in Git when there are multiple LCAs during a merge, but I think this is still better than the default.

```
Some stuff here
that is the leading context.
<<<<<<< HEAD
Only red kangaroos are supported.
||||||| merged common ancestors
Only orange kangaroos are supported.
=======
Both red and grey kangaroos are supported.
>>>>>>> kangaroo
More stuff goes here
that is the trailing context.
```

At this point, it is your (or Carol's) job to edit the file into a final version. In this case, Carol must inspect the rest of Bob's changes to see whether Bob's new claim about kangaroos is correct, but clearly the original text was wrong, and *one* of the changes should be kept.[15] If Bob's version is right, Carol should delete Alice's replacement line and the original line and the conflict markers, leaving Bob's change in place. If Bob's other change does not actually add support for grey kangaroos, Carol should delete Bob's replacement line, and the original text and conflict markers, leaving Alice's fix in place.

[15] Which new version is correct? I don't know, and neither do you. Carol may have to figure it out herself, but Bob is probably the best person to answer this question, and hence is probably the best person to do this merge.

Once Carol has the correct file in the work tree, she should run:

```
git add doc.txt
    or
hg resolve --mark doc.txt
```

To see which files still have unresolved merge conflicts, use `git status` or `hg resolve --list`. If you have started resolving a file and realize you have made a mess of it, you can restore the original conflicted merge—complete with conflict markers—using `git checkout --merge -- path` or `hg resolve path`. In Mercurial, you must run `resolve` with no flag *before* marking the file as resolved, or else first re-mark it as unresolved using `hg resolve --unmark`.

For instance, suppose Carol tries to resolve the conflict, but accidentally deletes most of the file while writing it back to her work tree, and then—thinking it is correctly resolved—runs `git add doc.txt` or `hg resolve --mark doc.txt`. Fortunately, before committing, Carol discovers her editing mistake. She can run:

```
git checkout --merge -- doc.txt
    or
hg resolve --unmark doc.txt
hg resolve doc.txt
```

and then re-edit `doc.txt`.

There is a way to get either Alice's or Bob's version of the file without having to edit out the conflict markers. *Be careful* when doing this: it is possible to discard changes you wanted to keep. Suppose,

for instance, that besides fixing the orange kangaroo, Alice fixed that "woombat", but Bob missed it. This change is therefore in the diff going from the merge base to the current commit, but not in the diff going from the merge base to Bob's latest. If you resolve the conflict by taking Bob's version of the file, you will lose the fix for the wombat. Nonetheless, this is a pretty handy trick, so here is how you do this with Git:

```
git checkout --ours -- doc.txt      Alice's file
git checkout --theirs -- doc.txt    Bob's file
```

Alice's file is "ours" because it is in the `HEAD` commit, and Bob's file is "theirs" because it is in the other commit.[16] This is how you do it in Mercurial:

```
hg resolve --tool :local doc.txt      Alice's file
hg resolve --tool :other doc.txt      Bob's file
```

Mercurial has an edge over Git for this particular case, because there are additional tools available besides just `:local` and `:other`, which we will get to in a moment, after we describe issues with automatic merges.

[16] User CommaToast suggested this, on StackOverflow, as a way to remember the ours/theirs distinction: "I guess since the head is the seat of the mind, which is the source of identity, which is the source of self, it makes a bit more sense to think of whatever HEAD's pointing to as being 'mine' ('ours', since I guess me and the HEAD makes two). If nothing more, that'll just be a good mnemonic device."

The VCS is stupid: its merge is purely textual

Note that *neither VCS understands anything* about the nature of the changes it is merging here. These merges are done strictly on the basis of the file text, broken into individual lines. This is not always suitable—for instance, merging XML-encoded data should probably be done quite differently—so both VCSes provide the ability to use arbitrary, user-supplied *custom merge drivers* (Mercurial calls these *external merge drivers*). Custom merge drivers can be difficult to write (depending on the task to be solved) and there are relatively few good examples of them. I found a simple one for Git at `https://gist.github.com/seanh/378623`; this handles changelog style files, merging them by treating them as insert-only and adding the inserted text from the other in front of the inserted text from the current commit.[17] (That is, in the case where Bob is merging Alice's changes, Alice's changes go at the front of Bob's changes. Furthermore, Bob's changes must occur at the front of the common base version, otherwise this merge driver stops, rejecting the merge.)

[17] This is, in fact, a specialized sub-case of Mercurial's `:union` merge tool. Git also has a union merge; we will see details later.

Both VCSes also provide a number of alternative built-in merge algorithms. Git calls these *strategies* while Mercurial calls them merge tools. Git has just a few built in strategies, called *resolve*, *recursive*, *octopus*, *ours*, and *subtree*. Mercurial has more built-in tools, all prefixed with a colon: *:dump*, *:fail*, *:local*, *:merge*, *:merge-local*, *:merge-other*,

:merge3, *:other*, *:prompt*, *:tagmerge*, and *:union*. We will see in a moment how to get Git's default `recursive` strategy to implement Mercurial's `:merge-local` and `:merge-other`. (Mercurial's *tagmerge* is marked experimental; I have not used it.)

Both VCSes also share a peculiar feature with regard to their various merge drivers: both first attempt each file merge using an extremely simple—trivial, really—algorithm: if a file is unmodified on one side, the VCS simply takes any modification found on the other side. Only if this trivial algorithm fails will they run the strategy-specific or custom or external merge driver. This is usually reasonable, since the most common case by far occurs when at most one side—current or other—has modified a file as compared to the base version. Taking a file straight from one side or the other is very fast as it is a simple copy operation. It also results in taking just one copy of the one change. However, if you wanted some special action (such as updating an internal date or counter) every time a file is merged, and put that action into your merge driver, it would not happen for these trivial merges. Mercurial provides a way to defeat the trivial merge, allowing you to enforce the use of your external merge driver. Git currently (as of Git version 2.12) does not: if the trivial merge succeeds, it will ignore any custom merge driver.

Merge strategies, options, and tools

Before we move on, we should explore Git's strategies and options, and Mercurial's built in merge tools, just a little bit more.

As we just saw, Carol can resolve a conflict in `doc.txt` by choosing either version (Alice's or Bob's) of that file. Since Carol merged Alice's changes first, and Carol also sees that Bob not only fixed the orange kangaroo but also added grey kangaroos, she might just take Bob's version. Alas, Bob did not notice the misspelled wombat. What if Carol could direct the merge to take Bob's version *only where there was a conflict*, and otherwise *combine* both Alice's and Bob's fixes? In this particular case, it's easy for Carol to do this by hand, but in a bigger set of changes, having the VCS do it for her would be a big improvement. Both VCSes can indeed do this. In Git's case, however, we need to start by knowing that this will be the right way to resolve the conflicts, so we will begin with Mercurial.

Mercurial's `hg resolve` easily allows us to retry merges, mid-merge, on a file-by-file basis, using a different Mercurial tool. Upon discovering that Bob's change should override Alice's, Carol need only re-run the merge of `doc.txt` using the `:merge-other` tool:

```
hg resolve --tool :merge-other doc.txt
```

This tool keeps Carol's current changes—which are just Alice's, really—wherever Bob's do not conflict, but takes Bob's changes wherever they do conflict. (Remember that Bob's is the "other" commit.) By taking Bob's changes, Mercurial automatically resolves the conflict and the file is now merged correctly.

With Git, Carol can achieve the same result, although she must choose it up front when running the initial merge, and then it applies to *every* file.[18] To do so, she need only add `--strategy-option theirs` to the initial merge command:

```
git merge --no-ff --strategy-option theirs kangaroo
```

This `theirs` strategy option has the same effect as Mercurial's `:merge-other` merge tool, i.e., it keeps our changes (which are copied from Alice's changes) when they do not conflict with Bob's, but keeps Bob's changes, discarding ours entirely, when they do.

As you might guess from the name *strategy option*, there are more strategy options. In fact, there are quite a few, and this is one case where Git's merge is slightly better than Mercurial's. The complete list is fairly long (and gets longer with newer versions of Git), but these are particularly noteworthy: `ours`, `theirs`, `patience`, and `rename-threshold=`*`threshold`*. The name "strategy-option" is long and tedious so from here on I will use the shorter spelling for this option, which is `-X`. That is, instead of `--strategy-option theirs`, we (or Carol) can write `-X theirs`.

We have already seen the `-X theirs` option; `-X ours` corresponds to Mercurial's `:merge-local` tool and simply chooses our change when there is a conflict. For Carol's merge, this would mean she would keep Alice's changes in favor of Bob's, although she would still pick up Bob's changes where there is no conflict. Carol has discovered that this would be wrong *for this one file, on this one merge*: In the one conflicting case, fixing the orange kangaroos, `-X ours` or `:merge-local` will keep the red kangaroo line, when we should keep the red-or-grey line instead. The problem here is that neither `-X` option or merge tool is *always* right: you, the operator of the VCS, must examine the conflict and determine which one (if any) is correct.

The `-X patience` option is short for the `-X diff-algorithm=patience` option. This uses a slower (more CPU-intensive) diff engine than the default `-X diff-algorithm=myers` diff. This CPU-intensive diff is more often able to notice and discount trivial or accidental matches.[19] The name *patience* is meant to imply that you will need more patience when using it, but if you get difficult-to-resolve merge conflicts, it may be worth trying.

The `-X rename-threshold` option sets the similarity threshold for Git's rename-detection. Remember that we said that Git must *guess*

[18] Git does include a command, `git merge-file`, that could fix this problem. This command can be used at any time, including in the middle of a conflicted merge. It is not designed for this use case, though, and needs a wrapper script to make it work properly.

[19] These occur often in source code with many blank lines, or lines consisting of just one open or close brace, as is common in a lot of computer languages. As computers get faster, this should perhaps become the default, but the time difference is still quite noticeable on large merges.

which files were renamed. By default, Git assumes that when file is at least 50% similar to another file with a different file-name, the second file came about by renaming the first file (and then maybe changing it somewhat). You can change this threshold to any other percentage. For instance, `-X rename-threshold=75` requires that the files be at least 75% similar. To see whether Git will detect a particular rename, you can run `git diff` with the `--find-renames=`*`threshold`* option; see Chapter XX for details.

Besides these `-X` options, `git merge` provides the `-s` *`strategy`* option.[20] Most of these are specialized enough for us to ignore here, but we need to call one out in particular because it is easy to misuse.

[20] This can be spelled `--strategy` *`strategy`*, but I find this is actually more confusing than just remembering `-s` and `-X`, with `-X` standing for "extended".

Confusingly, `git merge` provides `-s ours`, but `-s ours` has a *very different action* than `-X ours`. Git's `-s ours` corresponds to Mercurial's `:local` merge tool, whose action is to *ignore and discard every file from the other commit*, keeping the source tree the same as in the current commit. The principle use for this kind of merge in Git is to *kill off* a topic or feature branch, i.e., to merge its *history* back into the main-line branch (while ignoring its contents), then delete the branch-name entirely. This keeps the commits in the commit DAG for historical examination, while discarding from the main-line branch all the work that was done in the other branch, marking it as a failed experiment. (We can, of course, do the same in Mercurial, using `:local`. However, Mercurial's branches cannot be killed, so there is no real point to this.)

Note that Git has no strategy corresponding to Mercurial's `:other` merge tool. This merge tool is the symmetric opposite of `:local`, keeping every other-commit file while ignoring and discarding every current-commit (local) file while constructing the tree for the merge. Git does not offer a `-s theirs`, but it is easy to synthesize it (in fact, it is also easy to synthesize `-s ours` and Git probably should omit this strategy as it is too close in spelling to the very different `-X ours`). We will see how in a moment.

Git's `merge-file` command does offer the equivalent of Mercurial's `:union` tool, but again, `git merge-file` is too awkward to use directly. Mercurial's remaining built-in tools, `:dump`, `:fail`, and `:prompt`, have no corresponding equivalent in Git, but `:dump` is covered by Git's index, and `:fail` is only needed to force the use of an external merge driver (which, as we noted before, is not possible in Git). Mercurial's `:prompt` tool has little if any advantage over simply editing the version of the file containing conflict markers.

Why, and when, should we merge?

Although we have not yet covered rebasing, it is time to contrast merge and rebase. Without going into any detail yet, rebasing *includes* the source-combining aspect of merging, but *does not* record a merge in the commit graph. Rebasing works by *copying* commits, then throwing away the originals, in favor of the new copies. This is really the essential difference between the two. We'll see more about this in **??**.

Let's consider a more realistic merge sequence as well. Even if Carol *is* supervising both Alice and Bob, it usually makes more sense for the person who made a set of changes to integrate them. We saw this just a moment ago when Carol had to figure out whether Bob's changes actually supported grey kangaroos. It seems likely that Bob knows this offhand.

Let's also assume that Alice and Bob are actually working on different features. Perhaps Alice is working on wombats while Bob is working on kangaroos. (This would explain why Alice saw the "woombat" typo.) Alice and Bob might work in their own private branches, whose names we do not need to know.[21] They will integrate their work into the current development branch `dev`, probably through some more-central repository, perhaps by giving it to Carol when it is ready *and conflict-free*. To achieve this, Alice and Bob will need to rebase[22]—but in order to get their rebases right, they must know how to merge.

There is no hard and fast rule about when to rebase and when to merge. However, a small change—just a few commits—that can be reworked through rebasing, so that it looks like it was made after (and with full knowledge of) changes made by someone else, in a simple linear history, is often worth rebasing. This can make finding problems easier. On the other hand, if you have a long or complex history, or if you have published your commits and their hash IDs are now spread through many repositories. rebasing is probably unwise. You may introduce errors while adapting each copied commit to its new home, and even if not, you must get *everyone else* who has those commits to switch from the old, dull versions to the shiny new copies.

If you have a line of development that will take a long time or involve many people, and hence need to interact (i.e., re-combine) with other branches more than once, it's probably best to merge repeatedly. Each of these smaller merges allows the different developers to coordinate with each other while the changes—and the problems being solved—are still fresh in the minds of the people doing the work.

[21] We may wind up seeing them in Mercurial anyway.

[22] Technically, only one will have to rebase. Whoever is done first gets his or her commits added first; then the other must rebase.

Bibliography

Elaine B. Barker and Allen L. Roginsky. NIST SP 800-131a rev 1: Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Technical report, National Institute for Standards and Technology, November 2015. URL http://www.nist.gov/manuscript-publication-search.cfm?pub_id=919563. Supersedes SP 800-131A.

Michael A Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd edition, 2014.

Graham Cormode and S Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms (TALG)*, 3(1):2, 2007.

Quynh H. Dang. NIST SP 800-107 rev 1: Recommendation for applications using approved hash algorithms. Technical report, National Institute for Standards and Technology, August 2012. URL http://www.nist.gov/customcf/get_pdf.cfm?pub_id=911479. Supersedes SP 800-107.

Quynh H. Dang. FIPS PUB 180-4: Secure hash standard. Technical report, National Institute for Standards and Technology, August 2015. URL http://www.nist.gov/manuscript-publication-search.cfm?pub_id=919060. Supersedes FIPS 180-3.

Peter Heywood. The quagga and science: What does the future hold for this extinct zebra? *Perspectives in Biology and Medicine*, 56(1): 53–64, 2013. DOI: 10.1353/pbm.2013.0008. URL http://muse.jhu.edu/journals/perspectives_in_biology_and_medicine/v056/56.1.heywood.html. [Online: accessed: 2016-01-29].

J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison, 1975. URL http://www.cs.dartmouth.edu/%7Edoug/diff.pdf.

John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *Advances in Cryptology—EUROCRYPT 2005*, pages 474–490. Springer, 2005.

Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer Society*, 36(6):47–56, June 2003.

Jon Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596520123, 9780596520120.

Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986. DOI: 10.1.1.4.6927. URL http://xmailserver.org/diff2.pdf.

Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, Inc., 2009. ISBN 0596800673, 9780596800673.

programmers.stackexchange.com contributors. Empirical evidence of popularity of Git and Mercurial, 2014a. URL https://programmers.stackexchange.com/q/128851. [Online: accessed: 2016-01-05].

programmers.stackexchange.com contributors. Are there any statistics that show the popularity of Git versus SVN?, 2014b. URL https://programmers.stackexchange.com/q/136079. [Online: accessed: 2015-12-28].

Marc J. Rochkind. The Source Code Control System. *Transactions on Software Engineering*, 1(4):364–370, Dec 1975.

David SH Rosenthal. Keeping bits safe: how hard can it be? *Communications of the ACM*, 53(11):47–55, 2010.

Smithsonian Institution. Log book with computer bug, 1994. URL http://americanhistory.si.edu/collections/search/object/nmah_334663. [Online: accessed 2016-04-04].

stevemao. git log --tags changes the commits order, 2015. URL https://stackoverflow.com/q/34462011/1256452. [Online: accessed: 2016-03-18].

Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1, 2017. URL https:

//shattered.io/static/shattered.pdf. [Online: accessed 2017-03-11].

Walter F Tichy. RCS—a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology—CRYPTO 2005*, pages 17–36. Springer, 2005.