Practical Versionning with Git

Guille Polito, Stéphane Ducasse

July 10, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Expert Git

1.1 Some Git Internals

Before going on with the reproducibility concerns that brought you here to read this chapter and even before continuing with practical Git commands, we will dive a bit into Git concepts. Understanding a bit how Git works is useful when doing some more complicated stuff such as merging and branching. If you already know what is a Git commit, a Git reference and how the graph of Git objects is managed, you can skip this section.

Dissecting a Git Repository

Before starting explaining what is a commit, what is a branch, and so on, let's start easy by understanding the parts that compose our Git repository. When you create a Git repository as we did in the last section, or you clone an old repository that already has some files in it, you will find that there is more than meets the eye. A Git repository has usually three core collaborating components: the working copy, the repository, and the remotes. You can see an schematics on Figure 1-1.

What you usually see in your disk when you clone is not actually the Git repository but the **working copy**. The working copy is the directory where your files are, where you work and apply modifications. It is called a working **copy** because what you see is actually a copy of what is in the repository. The working copy is a write-able copy: you can freely modify it, break it, add new things or remove things.

Actually, you can do whatever change you want in your working copy, that Git will not take it into account, at least not automatically. Once your changes are ready, you have to commit them into your repository to store them in

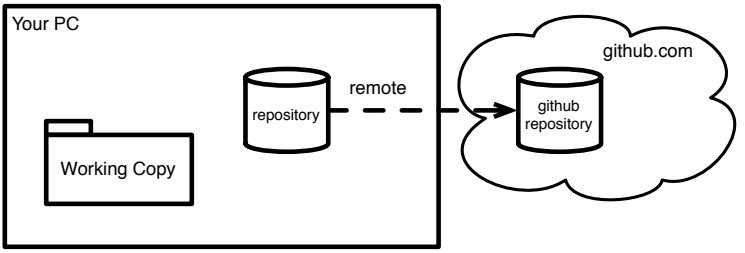

Figure 1-1 Git repository structure

your repository. A commit will take your changes, freeze them, and store them in the local database. Just for the curious ones, the local database (also known as **the BLOB** in the Git jargon) is stored inside your working copy, in a hidden directory called **.git**.

The commits you create from your changes live only inside your machine by default. If you want to share your commits with others, or to import commits from some fellow colleague, you have to interact with a remote repository (also called just remote). A remote is a distant Git repository that you will synchronize with your local one from time to time (this is where the famous pull and push come into play!).

Of course, this is an utterly simplified scenario. You could have a repository without a working copy. And your repository may have many remotes to synchronize with. But we will get into more complex stuff early on, no need to rush now.

A history-aware transactional database?

As we explained before, we usually work on the working copy, modifying our files and directories. Once we finished some work, we can freeze it and store it in the repository. That's what we call a **commit**.

From this perspective, a Git repository works as a transactional database. You are working on the changes of your disk, but they will not be effectively applied until you finish your transaction. Finishing your transaction is done, as in the database world, using the **commit** command. The result of this transaction is to create a new commit object in the Git repository. This commit object will contain an id (usually a hash such as `7ba52e5`) plus all changes we wanted to apply.

Git will store your last changes but also remember the entire history of changes you did. It keeps a list of all changes you did so you can do some nice stuff like for example:

- come back in time to recover some old change

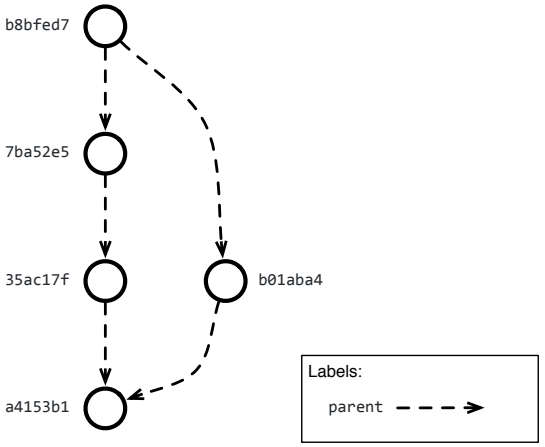

Figure 1-2 Graph of commits

- trace the changes in a file to see who (and why!) did a change
- analyze your repository and do some archeology, to see how your project evolved

It's a just graph of commits

The history of commits we explained before is not stored in a list form but in a graph form. A commit is a node connected to other commits by **parenthood**. A commit is said to be parent of another commit if it is the exact previous version. In other words, when we create a new commit, the parent of our new commit is the previous commit. A commit is said to be an ancestor of another commit if it preceeds it in history. Moreover, a commit can have one or many parents, and many commits can have the same commit as parent.

For instance, take a look at the schema of a typical commit graph represented in Figure 1-2.

- Commit `a4153b1` is the first commit in the graph, with no parents. A commit with no parents represents the first commit in a repository, when no previous history was available.
- Commit `35ac17f`'s parent is `a4153b1` and commit `7ba52e5`'s parent is `35ac17f`.
- Commit `b01aba4`'s parent is also `a4153b1`.
- Commit `b8bfed7` has two parents: `7ba52e5` and `b01aba4`.

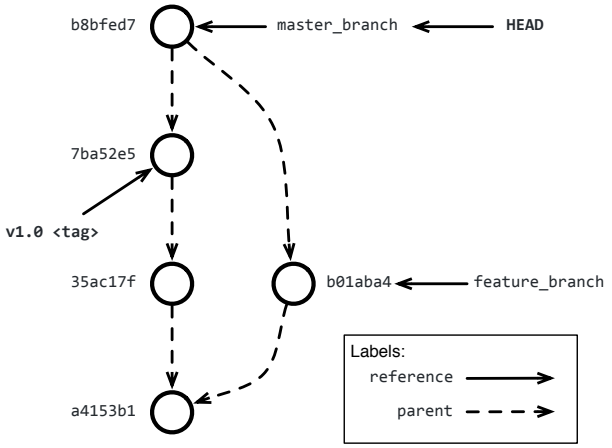

Figure 1-3 Git references

You may be asking yourself how can we arrive to such a situation. In short, a commit that is parent of many commits is creating an alternative history line: it is the result of a **branch** operation. Likewise, a commit that has many parents is joining two histories: it is the result of a **merge** operation.

Naming commits with references

You probably noticed that referring to commits by their id is awkward. Commit ids are generated automatically as hashes that avoid duplications as much as possible. However, they are not handy to work on a daily basis since they are hard to remember and type.

To solve this, Git provides a second kind of objects: Git references. A Git reference is like a label that you put on a commit, to be able to identify that commit by a much much simpler name afterwards. For example, you can name a commit as **release 1.0** or you can name it as **current development commit**.

As we show in Figure 1-3, there are two main kinds of references in Git:

- **tags**: tags are fixed labels that once created are not meant to be removed or moved. They are useful for doing releases: people will expect that a release does not change, otherwise they cannot depend on it.
- **branches**: branches are transferable labels that can be moved from commit to commit. They are used to maintain the different history lines of your project.

Another special reference, called **HEAD** is internally used by Git to know what is our current working branch. While it would look like an implementa-

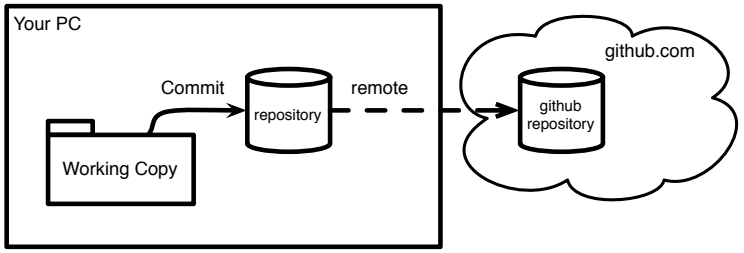

Figure 1-4 Commit is an operation that stores things from your working copy into your local repository

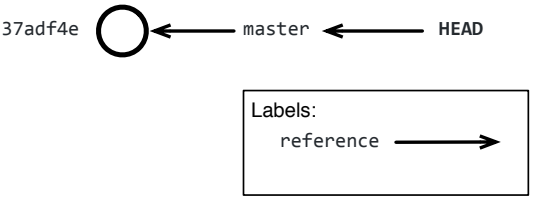

Figure 1-5 History graph after our first commit

tion detail, knowing that **HEAD** is there can save you many headaches as we will see later.

Now that you have built some strong conceptual Git muscles, we can continue in the next sections with some practical Git. Do not hesitate to come back to these sections to refresh some of the basics. As with any sport or discipline, understanding and practicing the basics is really important, since everything else is based on them.

1.2 Commit in workflow

We have already studied the commit operation ans saw that it moves changes from our working copy to our local repository, as it is shown in Figure 1-4. In additio to this staging view, we can see what the commit operation does from a graph point of view: The commit command creates a new node in our history graph. It then updates the master branch label to point to this new commit. The commit graph in this case will look as in Figure 1-5.

If we repeat the process, i.e. we apply a change to one of our files, add and commit our commit graph will change again. A new commit with a new commit id will be created having as parent our previous commit. The master branch label will be updated and point to this new commit. The commit

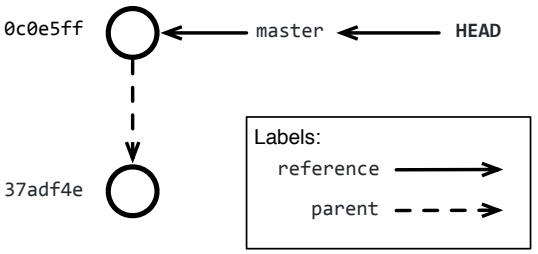

Figure 1-6 History graph after our second commit

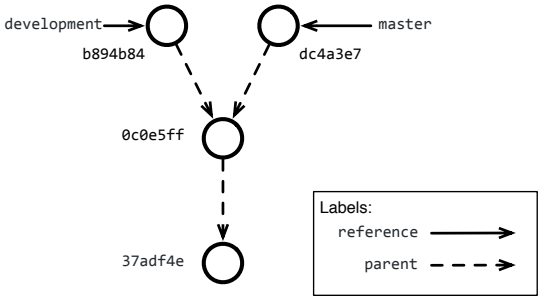

Figure 1-7 History lines can be branched from a commit

graph in this case will look as in Figure 1-6 Notice how our old commit is still there, but he's accessible as the parent of our new commit.

```
$ git add README.md
$ git commit -m "Adding a title"
[master 0c0e5ff] Adding a title
 1 file changed, 1 insertion(+)
```

1.3 Creating new history lines with branches

Branches in Git represent different histories. As in one of science fiction time-travel theories, Git branching is equivalent to take one moment in time have several alternative time-lines from there. Figure 1-7 illustrates the idea, showing that you can have two different futures from commit _0c0e5ff_.

By default, a Git repository will include a single branch, called **master**. Most people only need a single branch to work. However, it may be useful to split work in several branches as we will see later. You can ask Git for the branches in the repository using the command `git branch -v`.

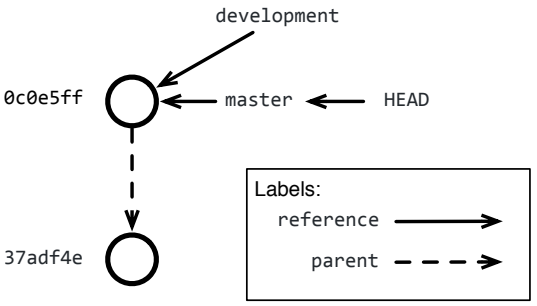

Figure 1-8 A new branch points by default to the same commit as the current branch

```
$ git branch -v
* master  0c0e5ff Adding a title
```

This command shows all branches in the repository, one per line. Then, for each branch it shows what commit it points, and the comment on that commit.

Creating a new branch

To create a new branch, we can use the command `git branch [branch_name]` giving as argument the new branch name. This will create a new branch from our current commit, the one that can be resolved from HEAD. Figure 1-8 shows what happens in the graph view.

```
$ git branch development
```

However, as we see in the graph view, creating a new branch does not modify HEAD. Indeed, our current branch/commit did not move. We will observe the same in the command line, if we ask the list of branches. The branch master is marked with a star, indicating it is the actual branch. And both branches point to the same commit.

```
$ git branch -v
* master      0c0e5ff Adding a title
  development 0c0e5ff Adding a title
```

To start working on our new branch, we just need to use the same `checkout` command we used for tags.

```
$ git checkout development
Switched to branch 'development'
```

Or alternatively, we could have created our branch using the `checkout -b` command, which performs a `git branch` and a `git checkout` one after the

other. Useful since these operations are usually done together most of the time.

```
# Instead of branch and then checkout
$ git checkout -b development
Switched to branch 'development'
```

Then, doing some work and creating a commit will only modify our current branch and leave `master` as it was before.

```
$ touch somefile
$ git add somefile
$ git commit -m "added somefile"
[development b894b84] added somefile
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 somefile
$ git branch -v
   master      0c0e5ff Adding a title
 * development b894b84 added somefile
```

Diverging history

Now that we have done some work in a branch, we can make our branches diverge. We only need to checkout another branch, existing or new, and start working from there.

```
$ git checkout master
Switched to branch 'master'
$ touch someotherfile
$ git add someotherfile
$ git commit -m "added someotherfile"
[master dc4a3e7] added someotherfile
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 someotherfile
$ git branch -v
 * master      dc4a3e7 added someotherfile
   development b894b84 added somefile
```

This change will create two diffent history lines, as shown in Figure 1-9. One history line represented by the `master` branch, and another history line represented by the `development` branch.

1.4 Creating Tags

When your project is in a stable state, it is often good to freeze it and put a name to that version. That way, other users can load the frozen version using that well-known name, and also be sure that version will not change. Freezing a version is particularly useful to reproduce a piece of sofware. A frozen version can be reloaded exactly as it is right now but in some point in

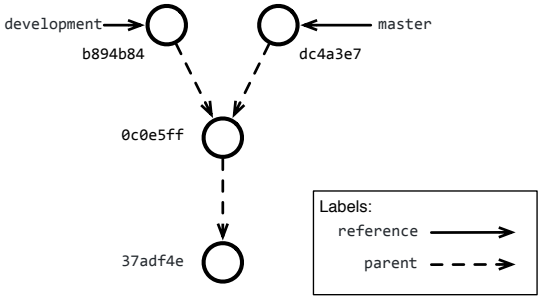

Figure 1-9 Divergent history

the future. Thus, software that depends on a frozen version can also benefit from its stability.

In Git, releasing is done via tags. A tag is a label that we put on a particular commit to be able to find it easily later on, so remember to put short, readable names to them. One particular consideration about tags is that they are not meant to be modified, although you will find in Git's documentation that you have special operations (that we do not recommend) to do that.

To create a tag, use the command `git tag` giving as argument a name for the tag and a descriptive message. Usual tag names use semantic version conventions, prefixed with a `v`. For example version `1` would be `v1.0.0`.

```
$ git tag -a v1.0.0 -m "First stable release"
```

You can afterwards list all your tags using the `git tag` command without arguments:

```
$ git tag
v0.1.1-alpha
v1.0.0
```

Finally, if you want to recover the code that you tagged at some point, you can use the `checkout` command with the name of your tag.

```
$ git checkout v1.0.0
Note: checking out 'v1.0.0'.

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:
```

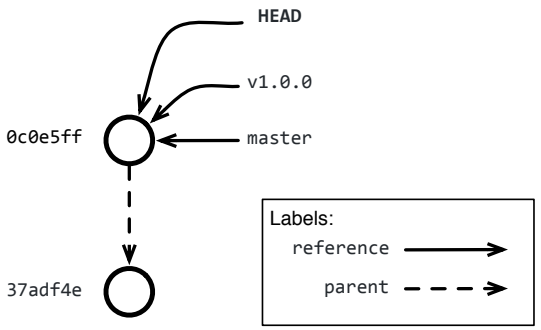

Figure 1-10 Detached HEAD after checking out a tag

```
  git checkout -b <new-branch-name>

HEAD is now at 0c0e5ff... Adding a title
```

When checking out a tag, Git tells you that we are in *detached HEAD* state. And that whatever commit we do in this state will be lost unless we create a branch. What happened here is that the `checkout` command modified the **HEAD** reference to point to the commit pointed by the tag, instead of a branch. Figure 1-10 shows the commit graph for this particular case.

1.5 Merging history lines

The most complicated part of Git is not branching or commiting, but merging. In our time-travel, time-line metaphores we said that branching is equivalent to open new time-lines. Merging is the equivalent to join them into a single history.

The concept behind merging is not difficult. Using the same idea of graph of commits that we used before, a merge can be represented as a commit that has several parents, thus joining several histories. Figure 1-11 illustrates such a merge commit.

However, as you see also in the picture, a merge commit will be referenced by one of the branches but not both. In other words, a merge operation means that a first branch will be merged into a second one. Thus the first one will remain intact. To perform a merge we need to checkout the branch that will host the changes, and then use the `merge` command with a branch name as argument. The following example shows how we can merge the development branch into the master branch.

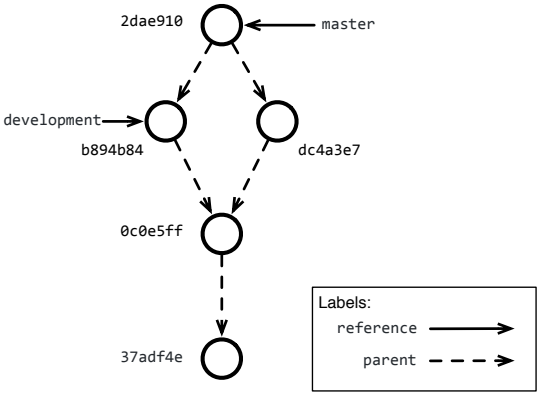

Figure 1-11 Merging the history with a merge commit

```
$ git checkout master
 ...
$ git merge development
[Merge made by the 'recursive' strategy.
 ...
 1 file changed, 0 insertions(+), 0 deletions(-)
 ...]
```

Managing Conflicts

When merging different history lines, things can go wrong if both history lines modified the same file or ressource. Such a problem is also called a conflict.

To understand the issue, let's generate a conflict on purpose. We can create two branches called `future-1` and `future-2` adding each the same file but with different contents:

```
$ git checkout -b future-1
$ echo "I'm in future-1" > conflicting.txt
$ git add conflicting.txt
$ git commit -m "Maybe will cause a conflict"

# Let's go back to master and redo the same in another branch
$ git checkout master

$ git checkout -b future-2
$ echo "I'm in future-2" > conflicting.txt
$ git add conflicting.txt
$ git commit -m "I'm sure it will cause a conflict!"
```

And then trigger a conflict when trying to merge:

```
# We are in future-2 so we will try to merge future-1
$ git merge future-1
Auto-merging conflicting.txt
CONFLICT (add/add): Merge conflict in conflicting.txt
Automatic merge failed; fix conflicts and then commit the result.
```

We see that as soon as we merge, Git tries to automatically merge the file `conflicting.txt`. It detects however a merge conflict that does not allow it to continue. If we check Git's status, you will now see:

```
$ git status
On branch future-2
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

  both added:      conflicting.txt

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

Git tells us that `conflicting.txt` is not merged and that we should fix it. To continue working, we should resolve such a conflict, telling Git what version we want to keep. Several solutions work: either we keep the version we had in **future-2**, we keep the version incoming from **future-1**, or we keep a can manually resolve the conflict and keep whatever version we want.

The easiest, non-thinking, way to merge is to open the conflicting file and resolve the conflict. For example, if we open our `conflicting.txt` file with a text editor we will see:

```
<<<<<<< HEAD
I'm in future-2
=======
I'm in future-1
>>>>>>> future-1
```

Git modified our file adding some `<<<<<<<`, `>>>>>>>` and `=======` markers in our file. What this markers delimit is the conflicts Git found. As the first line says, the first region (what is between the `<<<<<<<` and the `=======`) corresponds at the version that was in **HEAD** (i.e., **future-2**). As the last line says, the last region (what is between the `=======` and the `>>>>>>>`) corresponds to the version that was in **future-1**.

To resolve the conflict, you should: - remove all the special markers - keep only the version you want (or edit it to be different) - add and commit the conflicting file

For example, let's say we wanted to keep the version in **future-2**, we can edit the file leaving only

```
I'm in future-2
```

and then commit the resolved conflict:

```
$ git add conflicting.txt
$ git commit -m "Resolve conflict"
```

1.6 Interacting with Remote Repositories

So far we have worked only on the repository that resides locally in our machine. This means that mostly all of Git features are available without requiring an internet connection, making it suitable for working off-line (think on working on the train or with a constrained connection!). However, working off-line is a two-edged weapon: all your changes are captive in your machine. While your changes are in your machine, nobody else can contribute or collaborate to them. Moreover, losing your machine would mean losing all your changes too.

Keeping your changes safe means to synchronize them from time to time with a **remote repository**. A remote repository is a copy of your local repository that is stored remotely, that is, in somewhere else's machine. This could be, for example, in your company's or university's server, the cloud, etc.

In this section we will see how to interact with remotes, how to configure them, and how to synchronize our local repository with them.

Git Remotes

A Git remote is a Git server that is hosted in some machine other than ours. Usually, a remote will be hosted by some company like GitHub or GitLab, but it can be hosted also within our own company/university/research laboratory. Actually, we have already worked with a remote without knowing it, when we have cloned our repository in Section **??**. The code we used in that moment was:

```
$ git clone git@github.com:[your_username]/[your_repo_name].git
```

Which can be generalized as:

```
$ git clone [remote]
```

Once created, we can interrogate our repository for its remotes using the command `git remote -v`. We will then observe that git created automatically a remote named **origin** pointing to the location that we just cloned.

```
$ git remote -v
origin  git@github.com:[your_username]/[your_repo_name].git (fetch)
origin  git@github.com:[your_username]/[your_repo_name].git (push)
```

This first means that Git allows us to assign a name to avoid using urls all the way. In addition, we can see that Git differentiates remotes used for **fetch**ing from those used for **push**ing. Those differences are important for more advanced git configuration, that we will not cover in this chapter.

Adding and Removing Remotes

For advanced scenarios, when we need more than the default **origin** remote, we will need to use different remotes. All git commands interacting with a remote repository will have a variant accepting a remote repository as argument, as we will see later. In those cases, we can specify the remote's url on each of those commands to interact with the desired remote.

However, to avoid copy-pasting different remote urls all the time, Git provides us with the possibility of configuring new **named remotes** such as origin. The drawback of such an approach is that our list of remotes will need to be maintained from time to time, for example, if urls become invalid or our repository moves. In such cases, we will want to modify or remove old remotes to keep avoid errors or mistakes.

To create a new named remote we can execute the command `git remote add [remote_name] [url]`.

```
$ git remote add someRemote [url]
$ git remote -v
origin  git@github.com:[your_username]/[your_repo_name].git (fetch)
origin  git@github.com:[your_username]/[your_repo_name].git (push)
someRemote [url] (fetch)
someRemote [url] (push)
```

Existing remotes can then be renamed using the `git remote rename [old_name] [new_name]`. And in case the remote name you wanted to rename does not exist, Git will answer you with a falta error.

```
$ git remote rename someRemote company_remote
$ git remote rename non_existent newname
fatal: No such remote: non_existent
```

Existing remotes can then be renamed using the `git remote rename [old_name] [new_name]`. And in case the remote name you wanted to rename does not exist, Git will answer you with a falta error.

```
$ git remote rename someRemote company_remote
$ git remote rename non_existent newname
fatal: No such remote: non_existent
```

Finally, to remove an existing **named remote** you can use the `git remote remove [remote_name]`. And in case the remote name you wanted to rename does not exist, Git will answer you with a falta error.

```
$ git remote remove company_remote
$ git remote remove non_existent
fatal: No such remote: non_existent
```

Update your repository: Fetching and Pulling

Before being able to share our commits in some external server, we need before to update our repository to avoid them being out of synchronization. While you can always try to share your commits by pushing (see Section 1.6), you will see with experience that Git favors pulling before pushing. This is, among others, because in your local repository you can do whatever manipulation you want to solve mistakes and merge conflicts, while you cannot do the same in your remote repository.

Concretely, when using Git you have to have a state of mind where: 1. you update your repository 2. you fix **locally** whatever existing conflict between your work and the remote work 3. you then publish your changes.

> **Note** Actually, our recommended workflow has one more step before updating: commit. If you try to update when your working copy is dirty, updating can destroy your changes. Instead, if you commit before doing an update, your changes will be safely stored in the database. You'll be able to do any expert manipulation with your changes once they are in the repository.

As we said before, a Git repository is no other than a database. It is a database that stores commits and references to those commits. And to update this database, we require two basic operations:

- **fetch**. Bring the commits and references from a remote repository to your local repository without affecting your own.
- **merge**. Merge the remote references with your own references, the same operation explained in Section 1.5.

In addition, the **pull** operation does both fetch and merge in a single operation (Figure 1-12).

Fetching is done through the `git fetch [remote]` command, where we can specify both a remote url or a remote name as remote. Or, if we don't specify a remote, Git will by default fetch from whatever remote is specified as **origin**. Executing a **fetch** will show an output like the following:

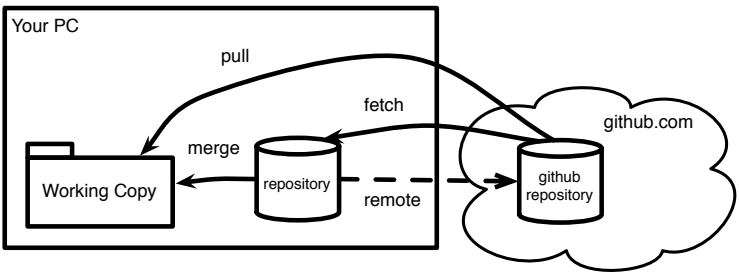

Figure 1-12 Fetch is an operation that brings things from a remote into your local repository. Merge will join the remote history with your current history and update your working copy. Pull will do both of them.

```
$ git fetch [remote_name]
remote: Counting objects: 79, done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 79 (delta 52), reused 74 (delta 52), pack-reused 4
Unpacking objects: 100% (79/79), done.
From git://github.com/[project_owner]/[your_repo_name]
   6b52ae6..5c53245  development -> [remote_name]/development
 * [new branch]      issue/876 -> [remote_name]/issue/876
 * [new tag]         v1.0     -> v1.0
```

Indeed, fetch will bring some objects (e.g., commits) to our repository, bring new branches and so on, but it will not update any of your branches or tags. We can then procceed to merge our local branch with the one in the remote by doing a normal merge operation but indicating a **remote branch** (that is, a branch prefixed by its remote name). Of course, as any merge operation, this can incurr into a conflict, that we should fix locally before continuing.

```
$ git merge [remote_name]/master
[Merge made by the 'recursive' strategy.
 ...
 1 file changed, 10 insertions(+), 1 deletions(-)
 ...]
```

These both operations could have been replaced by a `git pull [remote_name] [branch_name]` command. Pulling will fetch all commits from the branch named [branch_name] in the remote [remote_name] and then merge those commits with your current branch.

Share your commits: Pushing

The final step in our Git journey is to share our changes to the world. Such sharing is done by **push**ing commits to a remote repository, as shown in Figure 1-13. To push, you need to use the git command `git push [remote]`

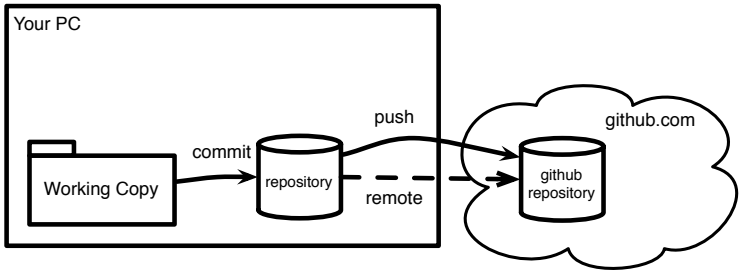

Figure 1-13 Push is an operation that sends commits from your local repository to a remote repository.

`[remote_branch]`. This command will send the commits pointed from your your current branch to the remote [remote] in the branch [remote_branch].

```
$ git push origin temp
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> temp
```

To avoid specifying the remote and destination branch on every push (which may be a bit verbose), you can avoid those parameters and rely on Git default values. By default the `git push` operation will try to push to the **branch's upstream**. A branch's upstream is the per-branch configuration saying to which remote/branch pair it should push by default. When we clone a repository, the default branch comes with an already configured upstream. We can interrogate Git for the branch's upstreams with the super verbose flag in the branch command, *i.e.*, `git branch -vv`, where we can see for example that our **master** branch's upstream is **origin/master**, while our **development** branch has no upstream.

```
$ git branch -vv   # doubly verbose!
  development   1656797 This commit adds a new feature
  master        f269295 [origin/master] First commit
```

On the other side, when a branch has no upstream, a push operation will by default fail with a Git error. Git will ask us to set an upstream, or otherwise specify a pair remote/branch for each push.

```
$ git push
fatal: The current branch test has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin test

$ git push --set-upstream origin test
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> test
```

Finally, another thing may happen while pushing: Git may reject our changes.

```
$ git push
To git@github.com:guillep/test.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to
    'git@github.com:[your_username]/[your_repo_name].git'
hint: Updates were rejected because the remote contains work that
    you do
hint: not have locally. This is usually caused by another repository
    pushing
hint: to the same ref. You may want to first integrate the remote
    changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
    details.
```

As the error message says, the remote has changes that we do not have locally, so we need to update our repository first. This can be solved with a pull (Section **??**) and a merge (1.5)

1.7 SSH or HTTPS access?

When cloning a repository or adding a remote, we will face the question of the URL: should we use HTTPS or SSH? Both HTTPS and SSH are secure ways of communication with remote machines. The main responsibilities of these communication protocols are: authentication (i.e., ensure that the two parties communicating are who they say they are) and secure transport through data encryption.

Although there are technical differences between the two, and both are capable of doing well their job, there is a main key difference between them: ease of setup, particularly for Windows users. Most Git repository hosts recommend using HTTPS because of this reason, and we do so as well.

We leave for the reader the task of investigating more about the differences between these protocols and setup SSH if he wants to.

1.8 Exercises

1. **Exercise 1**. Get a repository with many commits and checkout the parent of the current commit. This will put you in "Detached HEAD" state. Solve it using a new branch.

1. **Exercise 2**. Try to merge your previous branch into your new branch. What kind of merge is it?

1. **Exercise 3**. Repeat the scenario of the first exercise, apply a change to your one of your files and commit it. Try to merge your previous branch into your new branch. What kind of merge is it?

1. **Exercise 4**. Create a new online repository and push your changes into it.

1. **Exercise 5**. What is the smaller set of steps you could imagine to create a conflict?