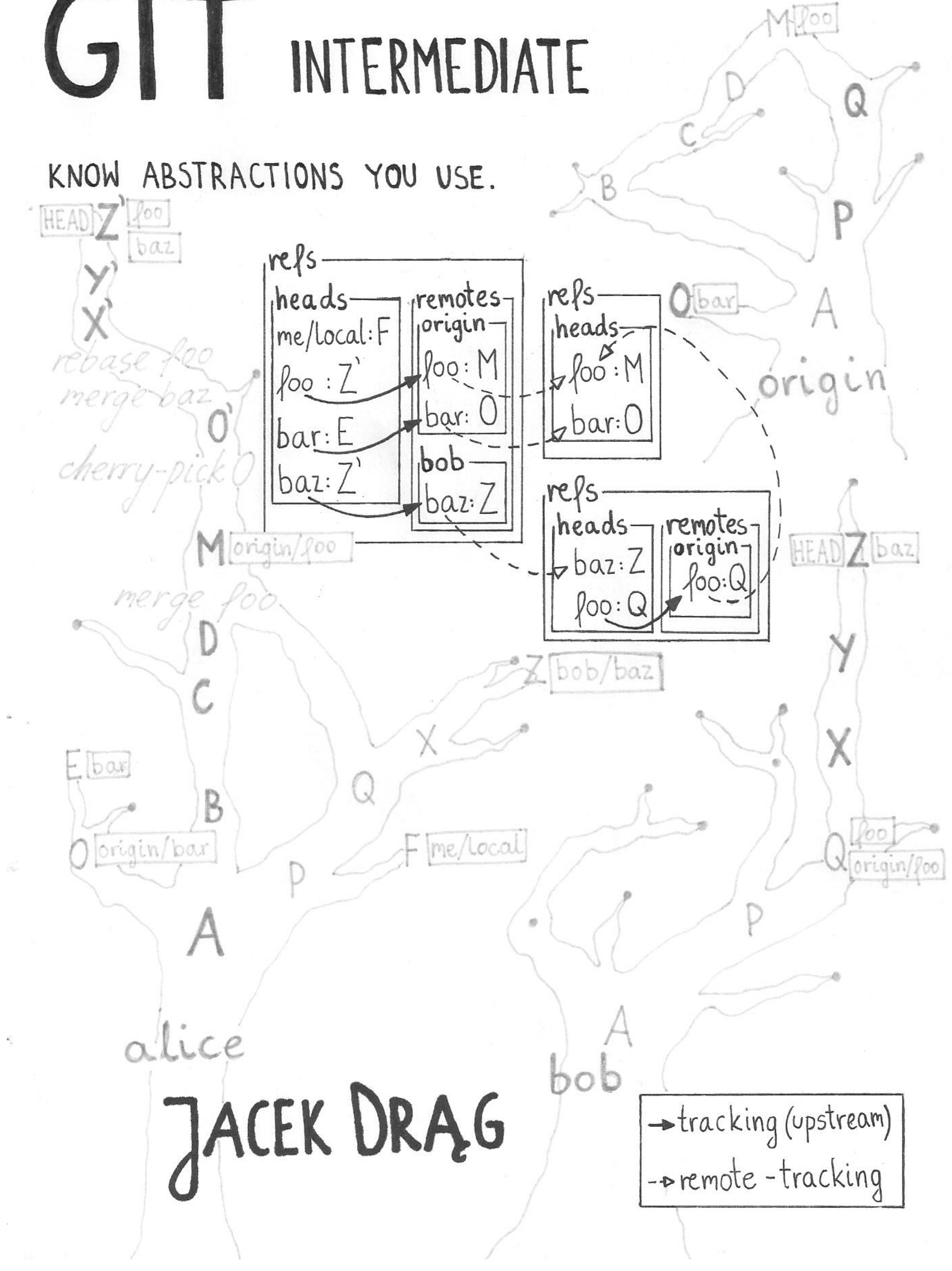
GIT INTERMEDIATE

KNOW ABSTRACTIONS YOU USE.



JACEK DRĄG

Git intermediate

Know abstractions you use.

Jacek Drąg

GIT intermediate
Know abstractions you use.
SAMPLE

Preface

Discover the beauty and elegance of Git! Learn to create decent, professional repositories!

Bearing the **proper abstractions** in mind, you know **why** and **what** should be done; It is easier then, to find the answer to **how** to do it.

The book gives the reader **a good sense of Git's mechanics**. It is particularly useful when less obvious situations occur.

The book can be read as "from zero to hero" guide. Nevertheless, after months/years of using Git, it is practical to take a break and return to the roots, from time to time.

The book is addressed mainly to people working with code and affecting the way the code is managed: programmers, devops, technical leaders, architects, project managers, etc. It may also be useful for students of technical faculties, or even for everyone who wants to have multiple versions of the work they create.

Git locally

Refs — normal and symbolic, branches

Now we know that Git repository is represented by directed, acyclic commit graph. Next goal is to:

- Expand (grow) the graph by creating new commits.
- Traverse the graph.

A *ref* points to a commit. (For programmers: analogy to the pointers is accurate). Refs are constantly used during everyday work. Ref, in opposite to commit, **can** change — first pointing to commit **A**, later to commit **B**.

There are two kinds of refs:

- *Normal ref* points to a commit. It just stores a commit ID.
- *Symbolic ref* points to another ref. It stores the name of the ref.

Most refs are stored in subdirectories of `.git/refs` directory. Some special refs (used for current work) are stored directly in `.git` directory.

One of the most important kind of refs is *branch*. By default, `master` is the first branch created during initialization of the repository.

Facility of creating, merging and distributed sharing branches is one of the fundamental Git features, that makes Git so powerful *distributed version control system* (*DVCS*).

Refs are used for effective dealing with the commit graph.
They can point directly to commits (*normal refs*) or to another refs (*symbolic refs*).
Branches are normal refs, so branches point to commits. These commits are called *tips of the branches* (from a given moment).

The most important ref — *HEAD*

HEAD is the most important ref. Git repository cannot exist without HEAD. Working with Git always means using HEAD, even if unconsciously.

Usually HEAD is a symbolic ref (it points to a branch. However, that is not always the case. When HEAD isn't a symbolic ref, you are in *detached HEAD* state (which will be discussed soon). Although some people like to work in detached HEAD state, it isn't a typical approach. In fact, if you work in detached HEAD state, Git will most likely be reminding you about it, and it will ask you to switch to the more typical state, which is the one with HEAD pointing to some branch.

HEAD

- Is stored in `.git/HEAD` file.
- Typically, if the repository is empty (no commits have been created), it points to the yet not existing branch `master`.
- During commit:
    - At first, the commit to which HEAD is pointing to (most likely — as a symbolic ref — indirectly), becomes the parent of the commit **being created**. That means, it is written to the one-element

parents list of the new-created commit.

  - After saving the new-**created** commit in the repository (map of objects), HEAD starts to point to this new commit. Normally it means, that the branch pointed by HEAD changes. In detached HEAD state however, this means that HEAD itself changes.

Git cannot function without **HEAD**. During committing, the commit pointed by HEAD becomes the only parent of the new-created commit.

A couple of experiments with refs and HEAD

Assuming that Git:

- Stores objects in `.git/objects` directory.
- Stores refs in `.git/refs` directory.
- Must have HEAD, which is stored in `.git/HEAD` file.

Let's try to create a Git repository manually.

Manual, minimalistic `git init`

```
mkdir experiment6
cd experiment6
mkdir -p .git/objects
mkdir -p .git/refs
echo 'ref: refs/heads/master' > .git/HEAD
git status
```

It works!

Let's keep going!

```
git symbolic-ref HEAD ①
git rev-parse HEAD ②
git commit --allow-empty -m"init repo"
touch y.txt
git add y.txt
git rev-parse HEAD ③
git commit -m"Committed when HEAD was $(git rev-parse HEAD)."
git rev-parse HEAD ④
git cat-file commit HEAD ⑤
```

① HEAD is a symbolic ref pointing to the branch `master` (we'll talk about `heads` in a while).

② HEAD is not pointing to any commit yet. There are no commits in the repository for now.

③ HEAD value before committing.

④ HEAD value after committing.

⑤ The parent of the new commit is the previous HEAD's value.

Heads as tips of the branches and branches

As mentioned before, the idea of branches is one of the most important concepts that makes Git so powerful.

What are those branches?
Technically, they are just (normal) refs, living in the `.git/refs/heads` directory.
Why not in `.git/refs/branches` though?!
The answer is:

Technically, a branch is just a ref. However, to realize what the term 'branch' really means, we have to consider the ref as a **tip of the branch**. The branch is defined by its tip and the parent relationship.
So, we can see the 'branch' as a subgraph: consisting of its tip itself and all commits reachable by parent relationship.

Now, reversing the order, we can see the branch as something that starts at the first commit (the root) of the repository and expands, possibly diverging and merging, until it finally reaches its tip — the current commit.
Moreover, that branch still can grow — it's as easy as creating a new commit on its tip (i.e. when HEAD is pointing to the tip).

If we take this point of view, our refs will indeed be pointing to the **tips** of the branches. Therefore, `refs/heads` may be technically more accurate location.

During work, HEAD usually points to the 'current' branch (its tip), e.g. `.git/refs/heads/master`. It allows the branch to grow by creating a new commit on the tip of the branch.

Useful checks

```
git symbolic-ref HEAD ①
git rev-parse refs/heads/master ②
git rev-parse HEAD ③
```

① `HEAD` is a symbolic ref, it points to some branch, e.g. `refs/heads/master`.

② `refs/heads/master` is a normal ref, which points to some commit — the tip of the branch (its head).

③ `HEAD` indirectly points to the same commit as `refs/heads/master`
Yeah — if `HEAD` points to `refs/heads/master`, what other commit could it indirectly point to?

Naming conventions

Name of a branch can include slashes (`/`). Using it makes the names look like file paths or contained in namespaces. It might be a good idea to decide on some convention for big projects. E.g. dividing branches into some categories:

- `feature/*` from short-lived branches intended to develop features,
- `user/<user>/*` for individual programmers, so that they can have their own branches without polluting the "general namespace",

or anything else that seems useful.

Useful commands

git branch	Display local branches.

git branch -vv	Display branches taking into account tracking info.
git branch --all	Display local and tracking branches.
git show-ref	Display refs.
git symbolic-ref <name>	Display which ref `<name>` is pointing to.

Lightweight tags

Another kind of refs are lightweight tags. They are normal refs used to (temporary) tag commits. To tag commits permanently *annotated tags* should be used, which are not refs but Git objects (see [git-objects]).

Addressing expressions — traversing graph, sets of commits

Documentation: ***gitrevisions***

As you know, commits are identified by the commit ID (SHA-1, hash). Along with ancestors relationship they create acyclic, directed graph. Thanks to the refs (like branches, annotated tags), some of the nodes of the graph (commits) can be easily reached.

But is it possible to reach any other commits in any other way than by refs or commits ID?

Well, it is. Git allows the user to identify commits by some *expressions*. By these expressions, you can reference:

- commits
- any Git objects (see [git-objects]) — trees, blobs, tags
- sets of commits

Most often expressions identifying commits are used, but the others can be useful as well. Commands used for calculating the expressions values are:

- ***git rev-parse***
- ***git rev-list***

These are complicated in general. Here is the simplest form:

```
git rev-parse <expressions>
```

displays *revision*, that is SHA-1 of the object/objects specified by `<expressions>`.

In many Git commands branches names etc. are passed as parameters. In reality, most often appropriate expressions can be used. A branch name is a special example of the simplest expression.

Some commands can get as parameters expressions that might not be obvious for the user, such as pushing a blob, instead of a commit.

Expressions identifying a commit

Most often used and the simplest expressions are:

- branch names, tags
- HEAD

Below **rev** will stand for some expression addressing a certain commit.

More complicated expressions that are most often used, are those using operators:

- tilde: `~`
- roof: `^`

rev~[n]

References the `n`-th ancestor of `rev`. For the merge commits it walks along the main inheritance line, i.e. it always picks the first parent from the list.

`n` defaults to 0, for `n` = 0 it just references `rev`.

Some identities

- `rev~0` = `rev`
- `rev~1` = `rev~`
- `rev~~` = `rev~2`
- `rev~~2` = `rev~3`
- `rev~3~2` = `rev~5`

We are just summing the amount of tildes, and that is how we calculate the amount of generations for which we have to go backwards along the main inheritance line.

rev^[n]

References the `n`-th commit on the parents list of `rev`.

`n` defaults to 0, for `n` = 0 it just references `rev`.

Some identities

- `rev^0` = `rev`
- `rev^1` = `rev^`
- `rev^1^1` = `rev^^`
- `rev^1^2` = `rev^^2`

There is no summing here. Every single roof in the expressions above references another parents list.

Let's notice that — so to speak accidentally — we have:

- `rev~0` = `rev^0` = `rev`
- `rev~` = `rev^` = `rev~1` = `rev^1`
- `rev~~` = `rev^^`

- `rev~~~` = `rev^^^`
- etc.

However, if tilde is followed by a number (different from 0 and 1), let's say `n`, the `rev~n` expression means something totally different from `rev^n`.

One could say that tilde is going backwards vertically, always choosing the most right direction (in the graph drown by `git log --graph`). On the other hand, roof goes horizontally along the parents list.

Tilde appears to be easier in usage (it's easier to type `rev~5` than `rev^^^^^`). But roof must be used in order to turn from the main inheritance path.

Small experiment

```
git rev-parse HEAD HEAD~0 HEAD^0 HEAD~ HEAD^
git rev-parse master master~0 master^0 master~ master^
```

<refname>@{<n>}

These expressions (with `<n>` positive) are calculated based on *reflog* (see [reflog]). `<refname>` can be skipped. In that case, the current branch will be used. Most often `<refname>` is a branch name but, among others, HEAD can be used as well.

@{-<n>}

The negative values reference previous HEAD values (branches/commits).

```
git checkout @{-1}
git checkout -
```

[<branchname>]@{upstream}

References the *upstream* of `<branchname>`, provided it is set.
`<branchname>` can be skipped, then the current branch will be used.
`@{upstream}` can be abbreviated to `@{u}`, so `@{u}` is the shortest form referencing the upstream of the current branch.

@

`@` alone is an abbreviation for `HEAD`.

Expressions identifying other Git objects

The previous expressions reference commits. Other expressions can be used to reference other Git objects.

```
rev^{<type>}
```

where *type* is optional and defaults to `commit`. Other possibilities are `tree`, `blob` and `tag`.

Let's say, that there is tag *v1.1* in the repository. In that case, expression `v1.1` references this tag itself, and the `v1.1^{}` (that is `v1.1^{commit}`) expression references the commit decorated by the tag.

```
git show --name-only
```

```
git rev-parse master^{tree}
```

Versions of blobs and trees

<rev>:<path> — values stored in commits

This references the value (blob/tree) of file/directory `<path>` stored in `<rev>`.
For instance, `HEAD~:README` is content of file `README` stored in the 'previous' commit. `master:README` is content of that file stored in (the tip of) branch `master`.
How can you reference to the project main directory (that is how to calculate `HEAD^{tree}`) in that notation? It will be `HEAD:` (with empty `<path>`).

:[<n>:]<path> — values stored in the index

This references the file content (blob) of `<path>` stored in the index as version `<n>` (see [blob-versions-in-index]).
`<n>` defaults to 0. It is the current file content. Others possible values are 1, 2 and 3. They are used to mark merge conflicts.
Thus, `git rev-parse :README.md` references the current (not conflicted) *README.md* file content stored in the index.

Having the file/tree value, you can display its content using `git cat-file` command, e.g.

```
git cat-file -p HEAD:README.md ①
git cat-file -p HEAD~: ②
```

① *README.md* file content from the current commit.

② Project main directory value from the previous commit.

Expressions referencing commits ranges

Some commands don’t operate on single commits, but on commit sets (ranges).

<rev>

In that context `<rev>` expression means reflexive-transitive complement of the parent relationship. Meaning all commits that are reachable from `<rev>`, i.e. the ones reachable by the ancestor relationship. Let’s remember that a commit is considered to be reachable from itself.

^<rev>

References commits unreachable from `<rev>`.

<rev1>..<rev2> — two dots, 'missing commits'

One could say, that this expression references 'missing commits'. I.e. commits reachable from the `<rev2>`, but unreachable from the `<rev1>`. In other words, these from the commit `<rev2>` history, which are missing in the commit `<rev1>` history.
This is an abbreviation for the `^<rev1> <rev2>`. That range is used, among others, during the `git rebase` operation.

<rev1>...<rev2> — three dots, symmetric difference

I.e. commits reachable from `<rev1>` or `<rev2>`, but not from both.

In both cases (two and three dots) you can skip `<rev1>` as well as `<rev2>`. They default to HEAD.

[<range>, …]

Some of the expressions above you can put one after another, which means the common part of the individual ranges.

E.g. `git log rev1 rev2 …`.

Graphical tools — Git isn't a hardliner

This book focuses on working with command line. However, sometimes it is more efficient to use graphical tools, including IDE. This is especially true for comparing files contents and conflicts resolving.

It turns out that Git is not a hardliner. Even during work with command line, it allows to use graphical tools for some purposes! Specifically, `difftool` and `mergetool` can be configured. Personally, I usually use `mergetool` for conflicts resolving:

```
git mergetool
```

Example of mergetool configuration

```
git config --global merge.tool intellij-idea ①
git config --global mergetool.intellij-idea.cmd 'intellij-idea-ultimate merge $LOCAL $REMOTE $BASE $MERGED' ②
```

① Defining a tool used for conflict resolving.

② Configuration of this tool (how to run it).

IDE can be also very useful for commit preparing. Especially creating a commit from only some selected changes can be much easier with IDE.

Normally, after creating a new file in the working tree, IDE asks if it should instantly add the file to the index. It is a good idea. Similarly, when deleting a file, it is better to instantly delete the file from the index.

Index (*staging area*)

At the very beginning, let's settle some issues:

Are `the index` and `the staging area` indeed exactly the same thing?
Yes, they are two names of the **same** thing.

The index is stored in the *.git/index* file.

This topic is often mistaken even by experienced Git users. So let's emphasize the **proper** abstractions again.

The staging area (also called the index) is the project state that will be saved with the upcoming `git commit` command.

`git commit` **does not save content of the working tree but content of the index**!

Therefore, preparing a commit is putting the project files in the proper states into the staging area.

One can see an analogy to working with ACID, read committed database. When you have an **open**

transaction in the database, you are modifying its contents:

- Adding some rows.
- Modifying some others.
- Deleting some others.

You can do this in many steps, and each next step can modify the rows modified in the previous steps. You can even:

- Modify rows that you have added earlier.
- Delete rows that you have added earlier.
- Restore rows that you have deleted earlier (might be hard).
- Roll back **all** the changes.

And all these changes (new state of the rows that are involved in the transaction) are invisible for the others until the changes are committed. During the commit the new database state is being saved and becomes visible for the other transactions.

The index can be seen as a modified project state:

- not committed yet
- not visible for the bystanders yet
- still modifiable,

which during committing will be (as a whole) saved and exposed as a new commit.

The index also stores the information about merge conflicts, but it will be discussed later.

There is one more **very important** difference:

- In the database transaction, **only** the rows modified by this transaction are involved. In the meantime other transactions can change other (not modified by your transaction) rows (read-committed transaction). As a result, the state after commit doesn't have to be exactly the one from before the transaction with the changes applied.
- Staging area is the **whole** project content (serializable transaction).

After committing the created commit will contain the **exact** index content. While committing Git does not care about:

- what is in the working tree,
- what is in other commits,
- it makes no comparisons,
- it does not apply any patches,
- it takes the index content **as it is**. And **this exact** content is saved in the new commit.

One of the most common mistakes is thinking that it is the information about changes between the commit and its parent what is stored in the index.

I do want to convince you how it really works. Therefore, let's do some experiment with the ***git ls-files*** command. The command displays information about the files in the index and in the working tree.

Experiment

- Clone some small repository (so that there are no changes in the working tree).
- Execute the following commands and watch.

```
git status ①
git ls-files --stage ②
rm .git/index ③
git status ④
git ls-files --stage ⑤
git reset --hard ⑥
git status
git ls-files --stage
> zzz
git ls-files --stage ⑦
git add zzz
git ls-files --stage ⑧
git commit -m"Add zzz"
git ls-files --stage ⑨
```

① "nothing to commit"

② Listing the files in the index.

③ Deleting the index. Normally, never do that!

④ "Changes to be committed" contains all project files!

⑤ Because the index is empty! All the files were previously in the index but were deleted from there.

⑥ Restoring everything (especially the index) to the state from HEAD.

⑦ The index hasn't changed.

⑧ There is a new file in the index.

⑨ The index hasn't changed.

You can guess the meaning of the majority of the results displayed by `git ls-files`. Except for the mysterious zeros. This will be discussed with merge conflicts (see [merge-conflicts]).

From the `git status` documentation

> Displays paths that have differences between the index file and the current HEAD commit, (...). The first are what you would commit by running git commit; (...)

From the `git add` documentation

> The "index" holds a snapshot of the content of the working tree, and it is this snapshot that is taken as the contents of the next commit.

`git checkout` vs `git reset`

One can ask: As `git checkout` and `git reset --hard` both change HEAD, the index and the working tree, are they not the same? There is one subtle, yet important difference between them:

hard reset Does not modify HEAD itself, but the branch pointed by HEAD.

checkout Modifies HEAD itself, switching it to another branch.
It doesn’t modify the branch itself — neither the original, nor the new one.

The above illustrates intentions of the two commands:

hard reset Clears the foreground and prepares for resuming the development

- of the current branch
- starting from the indicated commit.

checkout Clears the foreground and prepares for resuming the development

- of the indicated branch
- from its tip.

Table 1. changes made by the commands

command	changes HEAD/branch	changes index	changes working tree
`git reset --soft`	branch	no	no
`git reset --mixed`	branch	yes	no
`git reset --hard`	branch	yes	yes
`git checkout`	HEAD	yes	yes

Git remotely

The first part of the book was about working with a local repository, i.e. about growing the commit graph.

The second part will treat about cooperating with the remote repositories, i.e. about sharing (downloading and sending) parts of the graph from/to other repositories. This part is surprisingly short. It turns out that most of the hard work is local.

Branches: local, remote, tracking, remote-tracking and upstreams

There are several kinds of branches:

- ***local branches***, those are **normal** branches — We have been working with them so far.
  They can be displayed with the `git branch` command.
- ***remote branches***, those are branches in remote repositories. Let's notice, that they are local branches in the remote repository, and only from our repository's point of view they are remote.
  Actually, this isn't any special kind of branch. Usage of the *local*/*remote* term just allows us to distinguish if a 'normal' branch is from the local repository or rather from any remote one.
  The remote branches can be displayed with the `git ls-remote [<remote>]` command. The result is a little bit richer than the one for the local branches.
- ***remote-tracking branches***, those are representations of remote branches in the local repository.
  Such branches correspond to the remote branches with the same names and store the values (commit IDs) of appropriate remote branches. The values come from the last synchronization time with the remote repository. Those branches are not created explicitly!
  Remote-tracking branches can be displayed with `git branch -a` command. They are the ones like `remotes/<remote>/<branch>`. So, such a branch is identified by two names:
  - the name of the remote repository,
  - the name of the branch in this remote repository.
- ***tracking branches***, those are local branches related to some remote-tracking branches. The remote-tracking branch plays the role of *upstream* in this relationship. Similar to remote branches *tracking branch* isn't a **name** of a special kind of branch. It is a **statement**, that an upstream has been set for this branch. Each local branch can become or cease to be a tracking branch, because its upstream can always be set or unset

There are two main kinds of branches:

- ***normal branch***
  - It is identified by `refs/heads/<branch>`.
  - It can be a branch in a remote repository (i.e. identified by `refs/heads/<branch>` in the remote repository), not in the local one.
    It is called then a ***remote branch***. In case of a local repository it is called a **local branch**.
  - It can have an upstream (a remote-tracking branch) set, which allows it to cooperate with the remote branch.

    It is called then a ***tracking branch***.
  - It is changing during the graph development, i.e. during execution of commands: `commit`, `checkout`, `reset`, `merge`, `rebase` and `revert`.

- ***remote-tracking branch***
  - It is identified by `refs/remotes/<remote>/<branch>`.
  - It stores the value of a remote branch.
  - It can be an upstream of a normal branch (making the normal branch tracking).
  - It is changing only during exchanging the subgraphs with the remote repositories, i.e. during execution of commands: `fetch` and `push`.

upstream is a property of a local branch establishing a connection between this branch and a remote branch.
The values of upstreams are remote-tracking branches.

It’s high time for some experiments.

1. Let’s try to switch to a remote-tracking branch:

   ```
   git checkout remotes/origin/<somebranch>
   ```

   We are in detached HEAD state, in the commit pointed by `remotes/origin/<somebranch>`.

   Let’s go back:

   ```
   git checkout -
   ```

2. Let’s do it otherwise.

   ```
   git checkout origin/<somebranch>
   ```

   Detached HEAD again. Git has found `<somebranch>` in `remotes`.
   Let’s go back:

   ```
   git checkout -
   ```

3. Third time lucky:

   ```
   git checkout <somebranch>
   ```

   Something different! Git said:

   ```
   Switched to branch 'release'
   Your branch is up-to-date with 'origin/release'.
   ```

   ```
   git branch
   ```

A new branch has been created:

```
git branch -vv
```

```
* <somebranch> 1867d2b [<remote>/<somebranch>] <commit message>
```

Git found the branch `<somebranch>` among the remote-tracking branches `remotes/<remote>/*`. On that basis Git created branch `<somebranch>` and set its:

- value to the commit pointed by `remotes/<remote>/<somebranch>`,
- upstream to branch `remotes/<remote>/<somebranch>`.

This new local branch `<somebranch>` is ready to cooperate with remote branch `<somebranch>` from repository `<remote>`. These branches are related, by the upstream of the local branch. The value of this upstream is remote-tracking branch `remotes/<remote>/<somebranch>`.

For example, let's say that you are co-working with others developing branch `foo`.

Local branch `foo` can be related to some remote branch, e.g. `origin/foo`, by setting the upstream.

The intention is clear: You want to share your work — developed in `foo` — with others, by sending new commits to the remote repository. From there, they can fetch those commits. If someone has pushed their commits to that repository, you can fetch them to your local repository.

Setting an upstream explicitly

Switching to a remote branch creates a local branch and sets its upstream. However, sometimes you may want to manage the upstreams manually.

Setting the upstream explicitly

```
git checkout -b <somebranch> <remote>/<somebranch>^{commit} ①
git branch --set-upstream-to=<remote>/<somebranch> ②
```

① Creating the local branch pointing to the same commit as the remote-tracking branch. Without setting the upstream.

② Setting the upstream.

The upstream can be unset with:

```
git branch --unset-upstream <branch>
```

command. While the remote-tracking branches are not removed manually, it can be done either with a special command, or during fetching the changes from the remote repository.

`git fetch` — fetching subgraphs from a remote repository

Documentation: ***git fetch***

In this chapter you will learn to fetch commits from a remote repository.

Reminder: a branch is just a non-symbolic ref pointing to some commit. Sometimes we say that the given commit is the tip of the branch or the *value* of the branch.

Fetching a single branch

You know, that the value of local branch `foo` changes during execution of `git commit` command. It also changes during execution of the following commands: `checkout`, `reset` (`mixed` i `hard`), `merge`, `rebase`, `revert`.
But when and how does the value of remote-tracking branch `origin/foo` changes?

Let’s assume that new commits have appeared in the branch `foo` in the remote repository. Which means, that branch `foo` has changed in the remote repository (where the branch is local). Which means it points to another commit than the corresponding remote-tracking branch in your repository points to.

During execution of `git fetch origin foo` the following things happen:

- Checking `foo`'s value in the remote repository (what a commit it is) and writing this information to `.git/FETCH_HEAD`.
- If the commit is absent in the local repository, it gets fetched from the remote repository. The commit itself and **all** necessary objects, that is the whole history of the commit.
- In the local repository remote-tracking branch `origin/foo` gets moved to the commit written in `.git/FETCH_HEAD`.

That’s it! Your local graph has grown, and branch `origin/foo` points to the same commit as `foo` in the remote repository again.
Notice, that local branch `foo` has not changed!

A real example:

```
git fetch origin main
```

```
remote: Enumerating objects: 1449, done.
remote: Counting objects: 100% (1137/1137), done.
remote: Compressing objects: 100% (442/442), done.
remote: Total 1449 (delta 501), reused 1079 (delta 484), pack-reused 312
Receiving objects: 100% (1449/1449), 529.77 KiB | 11.52 MiB/s, done.
Resolving deltas: 100% (517/517), completed with 144 local objects.
From https://github.com/spring-projects/spring-framework
 * branch                  main       -> FETCH_HEAD
   7700570253..a6cd8a78e2  main       -> origin/main
```

Let’s see, what the individual lines mean:

```
* branch                  main       -> FETCH_HEAD
```

Value of the remote branch `main` is written to `FETCH_HEAD`.

```
7700570253..a6cd8a78e2  main       -> origin/main
```

```
 remote-tracking branch `origin/main` pointed to commit 7700570253`,
and points to `a6cd8a78e2`, now.
```

- Lines with
  - `Enumerating` — Git enumerates objects to be fetched.
  - `Counting, …, Resolving` — Git fetches the enumerated objects, compressing them for the time of transport.

If there was nothing to fetch, the message would be much shorter:

```
From https://github.com/spring-projects/spring-framework
 * branch                  main       -> FETCH_HEAD
```

How long does such a fetch last? You can experiment, fetching some repository, e.g.

```
git init
git remote add origin https://github.com/spring-projects/spring-framework
time git fetch origin main
time git checkout main
git log --pretty=oneline | wc -l
```

The result is not bad, as for such a large repository:

Several seconds to fetch the full project history

```
remote: Enumerating objects: 582278, done.
remote: Counting objects: 100% (555/555), done.
remote: Compressing objects: 100% (351/351), done.
remote: Total 582278 (delta 111), reused 449 (delta 87), pack-reused 581723
Receiving objects: 100% (582278/582278), 158.27 MiB | 17.29 MiB/s, done.
Resolving deltas: 100% (285908/285908), done.
From https://github.com/spring-projects/spring-framework
 * branch            main       -> FETCH_HEAD
 * [new branch]      main       -> origin/main

real    0m15,718s
user    0m18,050s
sys     0m2,499s
```

Half second to switch to a branch

```
Branch 'main' set up to track remote branch 'main' from 'origin' by rebasing.
Switched to a new branch 'main'

real    0m0,553s
user    0m0,346s
sys     0m0,203s
```

Repository contains over 26000 commits

```
 git rev-list --all --count
 26162
```

After `git fetch <remote> <branch>` execution, in the local repository:

- The value of the **remote-tracking branch** `<remote>/<branch>` is set to the same commit as the value of the remote branch `<branch>`.
- The whole history of the commit (the subgraph) is contained in the local repository.

`git fetch` is a 'safe' operation. It modifies only:

- the local repository — adding new object to it,
- the remote-tracking branches,

and it does **nothing**:

- in the working tree,
- with the local branches (even with the tracking ones).

In addition, all tags reachable from the tip of branch `<branch>` are fetched.

Fetching many branches

You don’t have to specify the branch you want to fetch.

```
git fetch <remote>
```

is enough. Git will do a bit more, it:

- Fetches all new remote branches and creates remote-tracking branches for them.
- Updates all remote-tracking branches.
- Fetches tags reachable from the remote-tracking branches.

Usually, you don’t have to specify the remote you want to fetch from. You can type:

```
git fetch
```

How does Git know which remote repository should be taken into account? There are two options:

- If the current branch is a tracking one, the remote of the upstream is used.
- Otherwise, the name *origin* is used.

The changes from all remote repositories can also be fetched at once:

```
git fetch --all
```

In fact the above examples are the most common cases. In reality the Git world is much richer and a simple `git fetch` uses various configurable settings. As usual, the defaults allow the user not to care too much about it.

General form of fetching

In general, `git fetch` command looks like this:

```
git fetch [<options>] [<repository> [<refspec>…]]
```

Instead of simple `<remote>` more general `<repository>` can be used. Instead of `<branch>` a `<refspec>` list can be used.

We will go back to that after describing [repo-and-urls] and [refspec]. In particular, the following versions of the `fetch` command will be described:

```
git fetch <remote> <remote-branch>:<local-branch>
```

Deleting withered branches

A remote branch can be deleted, but it does not mean that the remote-tracking branch, which tracks the branch, will also disappear. It won't. Even after execution of `git fetch`. The only thing done when fetching is noticing in the local repository that the corresponding remote branch does not exist anymore. One could call a remote-tracking branch without its remote counterpart: dangling (as it is a ref) or withered (as it is a branch).

You can make `git fetch` delete withered branches this way:

```
git fetch -p
```

or without fetching with the command:

```
git remote prune <remote>
```

Notice, that both commands have to connect to the remote repository to see which branches were there deleted.

Useful options

--dry-run	Do not execute. Just show what is about to happen.
(-a | --all)	Fetch from all remotes.
--depth=<depth>	Do not fetch the whole history but to the given depth only. There exist also several similar options.
(-p | --prune)	Delete the remote-tracking branches which don't have the corresponding remote branches anymore.
(-P | --prune-tags)	Delete all tags absent in the remote repository. Including these which were created only locally!

(-n | --no-tags)	Do not fetch the tags.
(-t | --tags)	Fetch all tags, even not reachable from any branch. This option can be useful when the branch was deleted but some tags, not reachable anymore, are still relevant. Normally, only reachable tags are fetched.

`git fetch` modifies the remote-tracking branches **only**. It does not modify the local branches unless forced!

Supplements

`git rebase` — default values of parameters

`git rebase` command was described in the first part of the book (see [git-rebase-local]). But the second part was needed to fully describe `git rebase`'s default parameters.

In the general form below all parameters are optional:

```
git rebase [-i] [--onto <new-base>] [<upstream> [<branch>]]
```

If you don't give any of them, the following values will be used:

```
git rebase --onto HEAD@{upstream} HEAD@{upstream} HEAD
```

For example, if `master` is the current branch:

```
git rebase
```

expands to:

```
git rebase --onto origin/master origin/master master
```

This command rebases the commits added to branch `master` locally onto `master` fetched from `origin`. So it is integration of the local development with what has been developed remotely.

By default, the values of the command:

```
git rebase [-i] [--onto <new-base>] [<upstream> [<branch>]]
```

are:

`<branch>`	HEAD, more precisely the current branch.
`<upstream>`	Upstream of branch `<branch>`.
`<new-base>`	The value of the `<upstream>` parameter In particular, if parameter `<upstream>` is not given explicitly, `<new-base>` defaults to the upstream of branch `<branch>`.