

# git

## An Introduction

What is it?

Why should you use it?

How does it work?

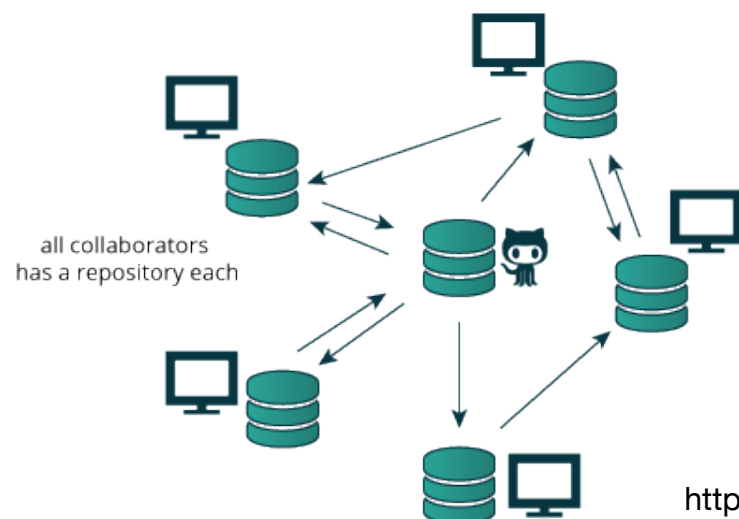
# What is it?

*“Git is a free and open source **distributed version control system**, which is fast and efficient.”*

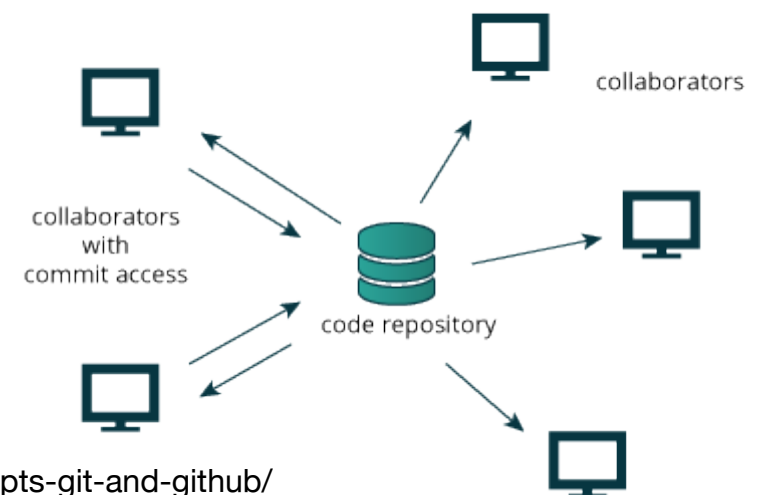
**- Git Homepage**

- Version control system = tracks versions of files  
e.g. source code, LaTeX thesis, paper or talk, website html, etc.  
(Rule of thumb: Everything you edit in text editors)
- Distributed = everyone has a full local copy of the repository

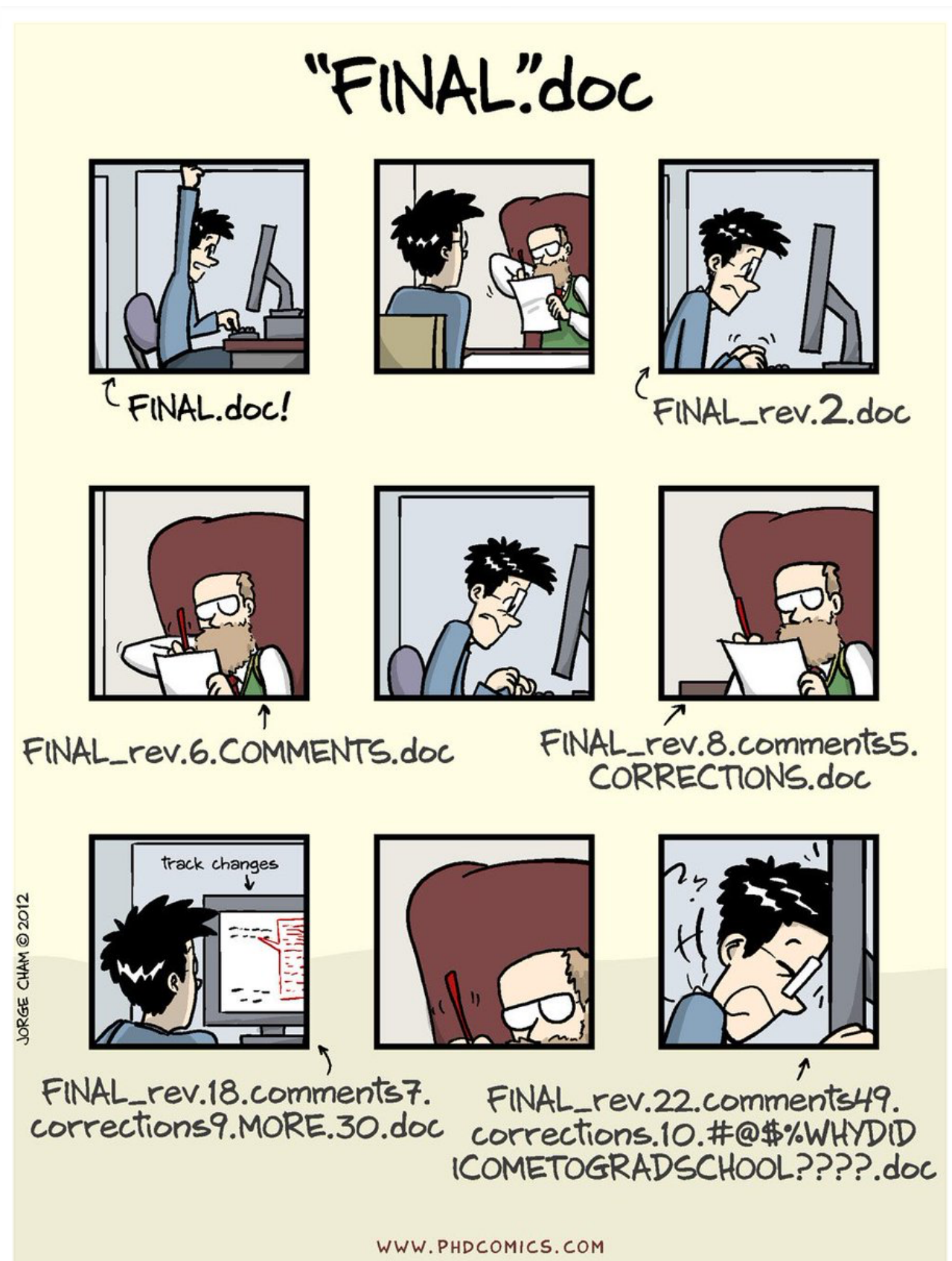
distributed



centralized



# Why should you use it?



# Why should you use it?

It helps you!

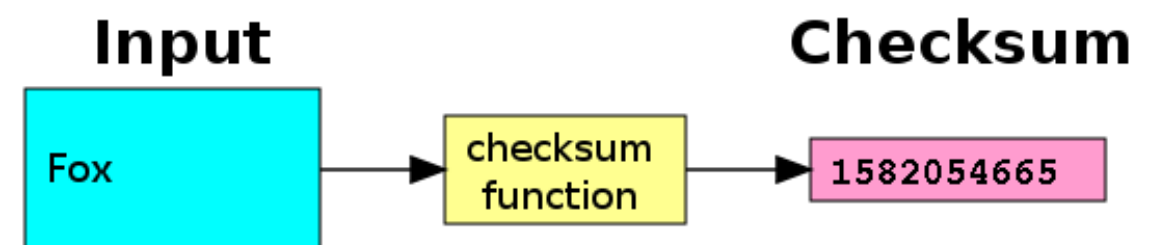
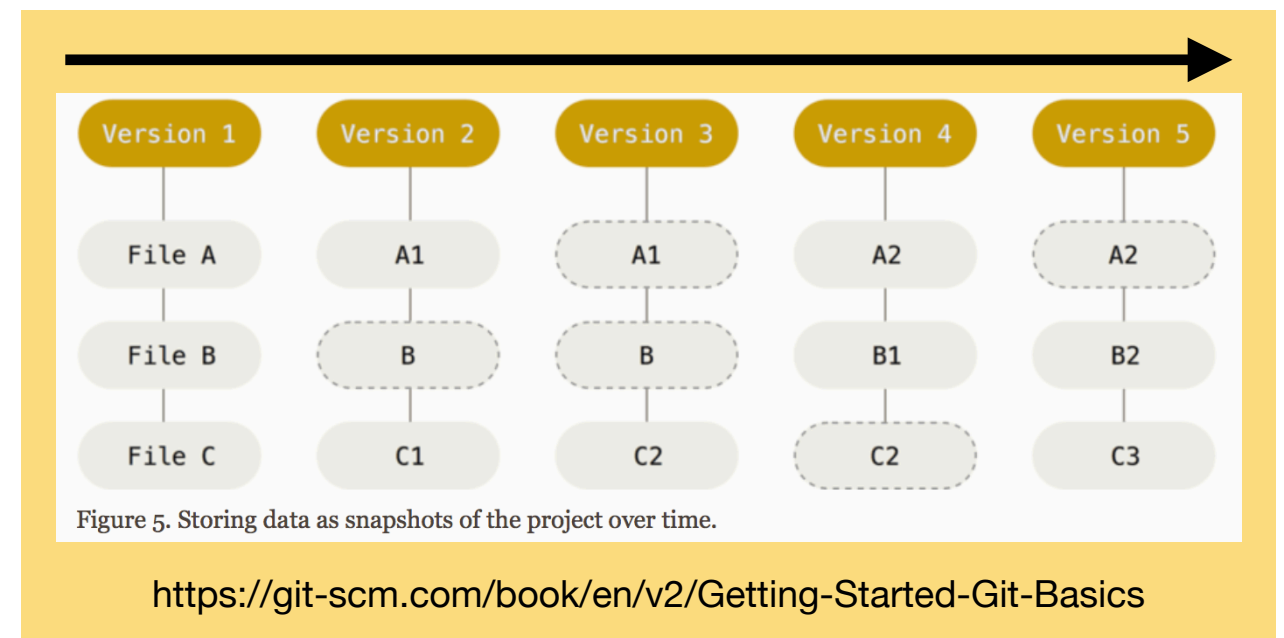
- Transparent history of all changes
- Moving back and forth in time
- Everything is easily traceable and reversible (e.g. errors)

Good scientific practices:

- Reproducibility and traceability
- Enables collaboration

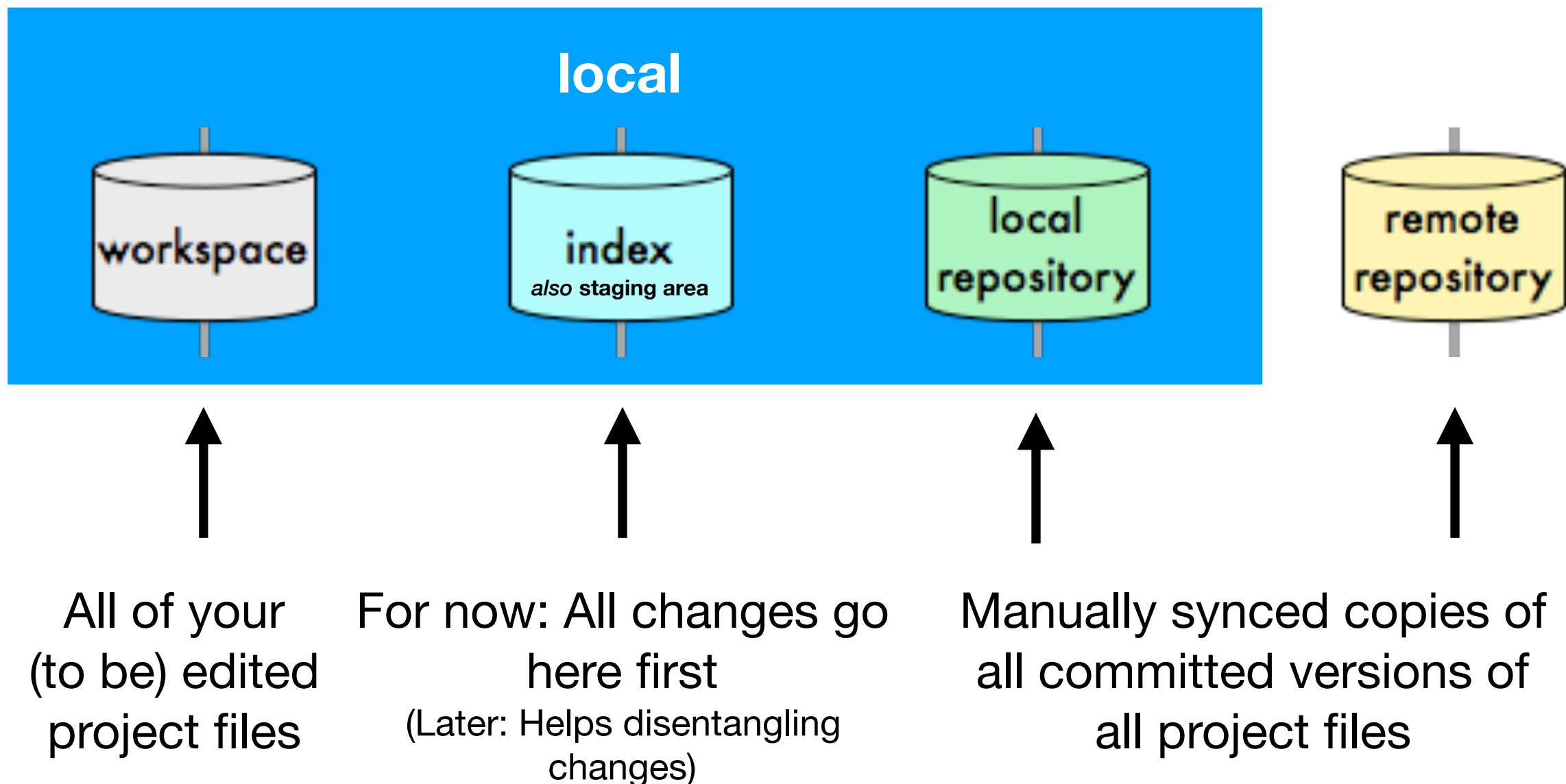
# How does it work?

- Repository = database containing all versions of the files
- Snapshot-based system
  - Snapshots are called **commits**
  - Commits are named by checksums (also used to ensure data integrity)
- Almost every operation is local
  - Working without network connecting
  - Distributed system (everyone carries a backup)



# How does it work?

## 4 distinct places



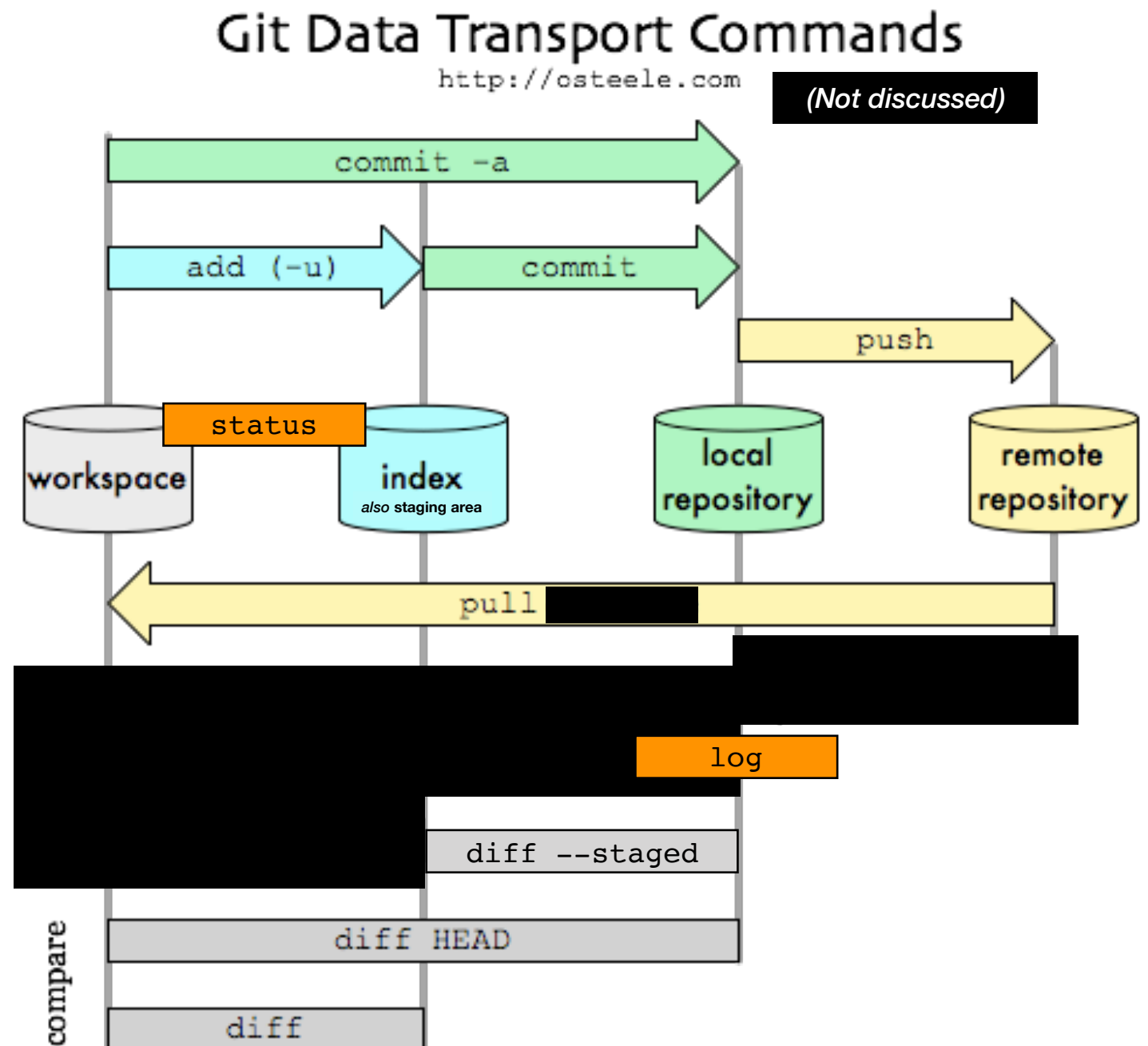
# How does it work?

- Overview of most common commands
- Most common use by command line

**Don't Panic**

**This is an  
introduction.**

**It is worth it.**



# Gist of this Introduction

## Command by command

1. The basic workflow: Set up a local repository and save changes to it  
—> Exercise 1
2. Branches: Work on different features in parallel  
—> Exercise 2
3. Basic interactions with a remote repository



# Notation

A white dollar sign (\$) on a dark gray square background.

what follows are commands to be entered on the command line e.g. `$ git log`

A white symbol consisting of an angle bracket, three dots, and another angle bracket (<...>) on a dark gray square background.

marks names and other variables that change during use e.g. `$ git branch <branch_name>`

## Advanced:

Tips for more advanced user will be in green boxes

```
$ git help
```

**Get help** for any command

```
$ git help <command>
```

```
$ git <command> --help
```

```
$ man git-<command>
```

Works offline!

**Installation:** Good changes it is already installed, if not [\[Link\]](#)

# Commands

**config**

**init**

**status**

**add**

**commit**

**diff**

**log**

branch

checkout

merge

clone

push

pull

```
$ git config
```

Before doing anything else: **Identify yourself**

```
$ git config --global user.name <your_name>
```

```
$ git config --global user.email <your_email>
```

Make sure this is consistent across all your machines

Make your life easier:  
Set up **tab completion**  
[\[Link\]](#)

### Advanced:

#### Aliases

```
$ git config --global alias.st „status“
```

„Did you mean?“

```
$ git config --global help.autocorrect 10
```

```
$ git init
```

## Initializing a empty (local) Git repository

```
$ git init  
Initialized empty Git repository in /path/to/  
example_repo/.git/
```

Adds a hidden `.git` directory, where git stores all of its information

Git only uses relative paths (source directory can be moved freely)

```
$ git status
```

Display **status information** of working directory and staging area

```
$ git status
On branch master

No commits yet

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

  hello.cpp

nothing added to commit but untracked files present
(use "git add" to track)
```

(Output for a newly initialized repo with one new file `hello.cpp`)

```
$ git add
```

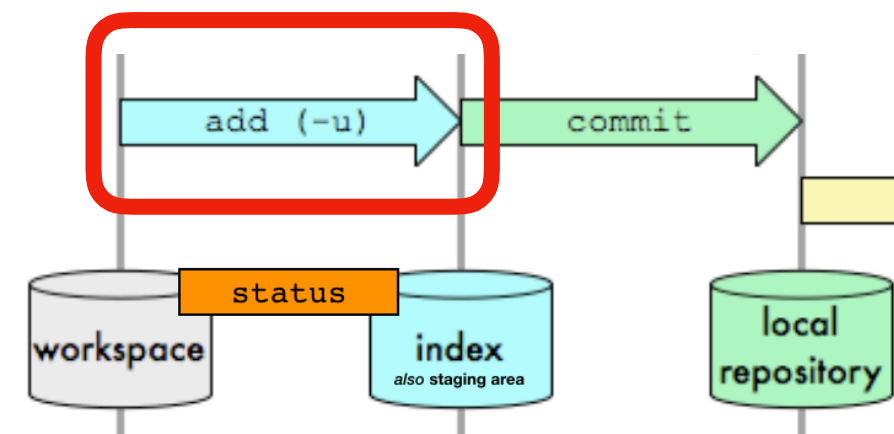
## Adding changes to the staging area

```
$ git add <file_name>
```

Adds all changes of a file to the staging area

```
$ git add <directory_name>
```

Adds all changes of all files in the directory to the staging area.



### Advanced:

You can also add only specific changes to a file (so called *hunks*) to the staging area.

```
$ git add -p <file_name>
```

Ignore certain files with `.gitignore` [\[Link\]](#)

```
$ git commit
```

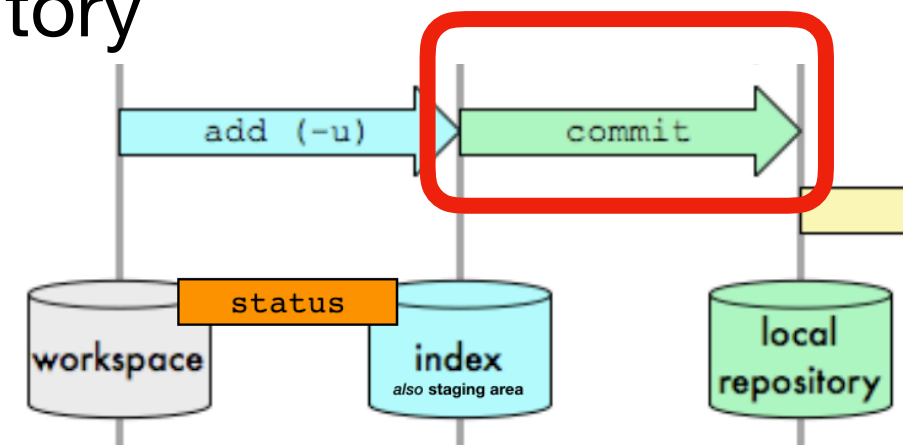
Committing staged changes to your repository

```
$ git commit
```

External text editor (most likely `vi`) will open and ask for a commit message

Don't like `vi`: 

```
$ git config --global core.editor „nano“
```



Invest in good commit messages!

- Subject line + body (Follow 50/72 rule [\[Link\]](#))
- Write them like a email to yourself / the other developers
- Document why you made the changes

**Good commits are small and often, conceptually separated, only include source files & at best working code**



```
$ git diff
```

**Display changes** to your tracked files

```
$ git diff
```

To be precise: Differences between working directory and staging area ➡ only unstaged changes

Helpful to inspect what you have done

```
$ git log
```

**Display history** of your commits

```
$ git log
```

See the last changes that were made  
including the commit message (at least per default)

# Exercise 1

**config**  
**init**  
**status**  
**add**  
**commit**  
**diff**  
**log**

branch  
checkout  
merge

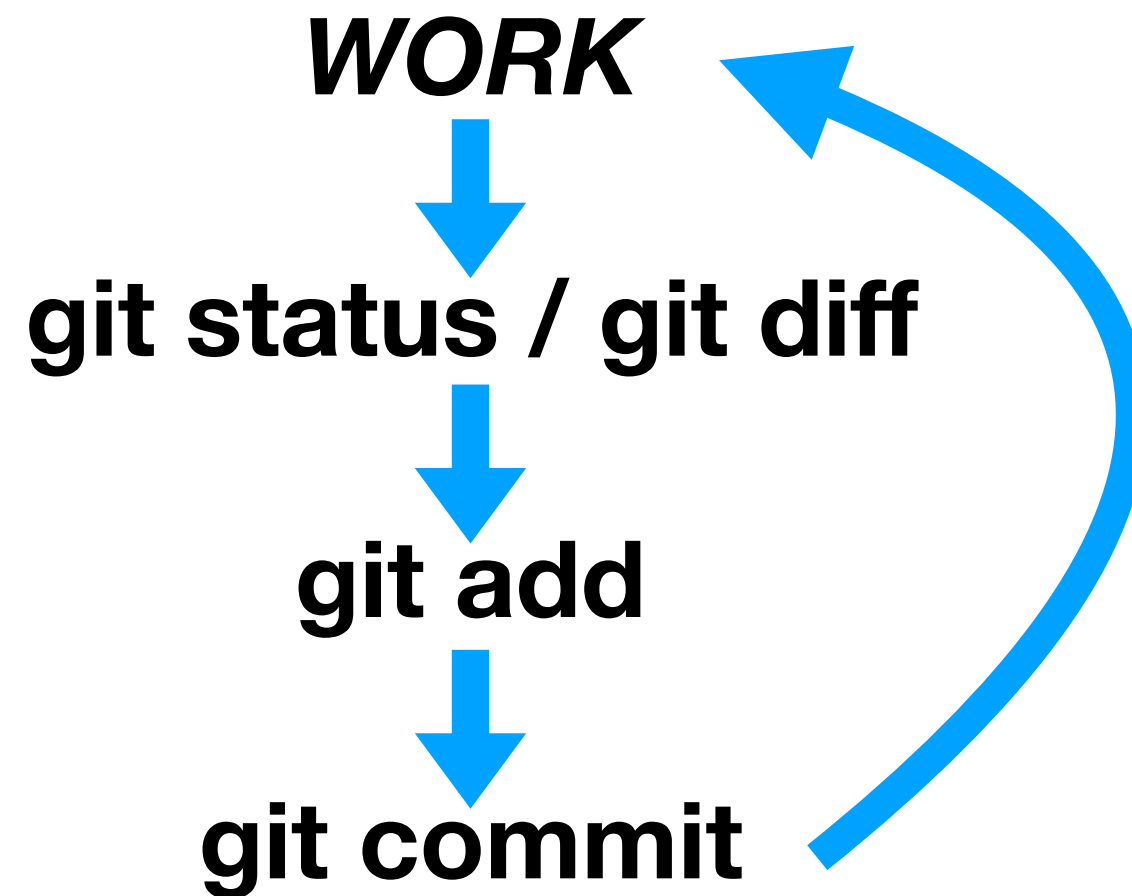
**Solutions**  
can be found at the  
end of the slides

# Exercise 1

1. Configure your git setup by setting your name and mail address. If you done that, check the `.gitconfig` file in your home directory.
2. Create a new directory and initialize an empty git repository in it.
3. Create a simple sample code file in the directory and commit it.
4. Modify the sample code file and commit the changes. Check the changes first.
5. Look at the commit history you created.

**Optional:** If you currently work on a "code" project (remember, this also might e.g. be a LaTeX paper project), repeat step 2 and 3 and make it git repository. Instead of creating a sample file, you just commit your source files. Do not worry, your project files will remain untouched by this. The next time you change your project files, just commit your changes (step 4 and 5). Just continue to repeat step 4 and 5 every time you work on your project and you will have already mastered the main git workflow.

# Basic Workflow



```
$ git commit -a
```

**Directly commit** unstaged changes

```
$ git commit -a
```

Shortcut for

```
$ git add -u  
$ git commit
```

**Advanced:**

Stage all changes to  
tracked files at once

```
$ git add -u
```

What is the purpose of the staging area?

- Allows to only commit part of your changes (*Assemble your commit to your liking*)
- Split changes across commits
- Also other use cases e.g. for reviewing your changes, ...

# Commands

config

init

status

add

commit

diff

log

**branch**

**checkout**

**merge**

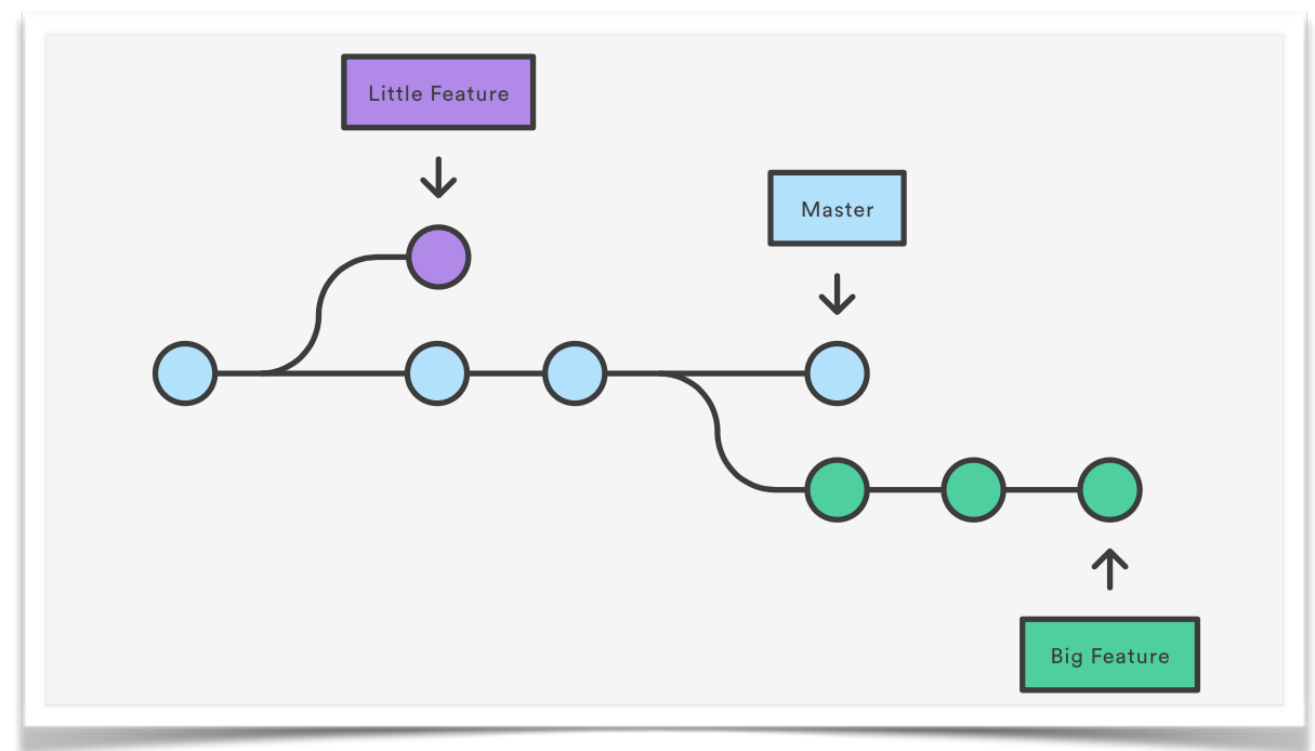
clone

push

pull

# Branches

- Branches store **different versions of your project**
- Parallel development
  - Implement new features
  - Fix bugs
  - Try out something
- Cheap to do in git  
(technically just pointers to a commit)
- Main branch = `master`
  - By default created at initialization
  - Usually development is done on other (feature) branches



<https://www.atlassian.com/git/tutorials/using-branches>



```
$ git branch
```

**Create** new branch

```
$ git branch <branch_name>
```

**List** all branches of local repository

```
$ git branch
```

**Delete** branch

```
$ git branch -d <branch_name>
```

Safe option to delete a branch, since it prevents data loss

```
$ git branch -D <branch_name>
```

Use CAREFULLY! Be sure you want to lose this progress

```
$ git checkout
```

for branches

**Switch** between existing branches

```
$ git checkout <branch_name>
```

Changes your project files

Only works with no uncommitted changes („Clean working tree“)

**Shortcut:** Create and checkout new branch

```
$ git checkout -b <new_branch_name>
```

# The HEAD

**HEAD** = special pointer to currently checked out branch (commit)

```
$ git checkout master
```

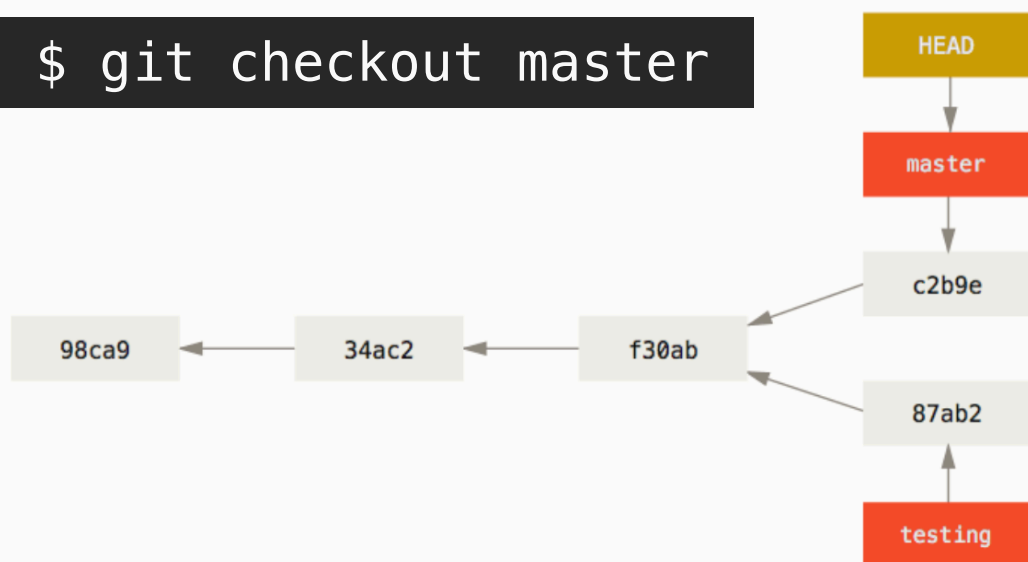


Figure 17. Divergent history

```
$ git checkout testing
```

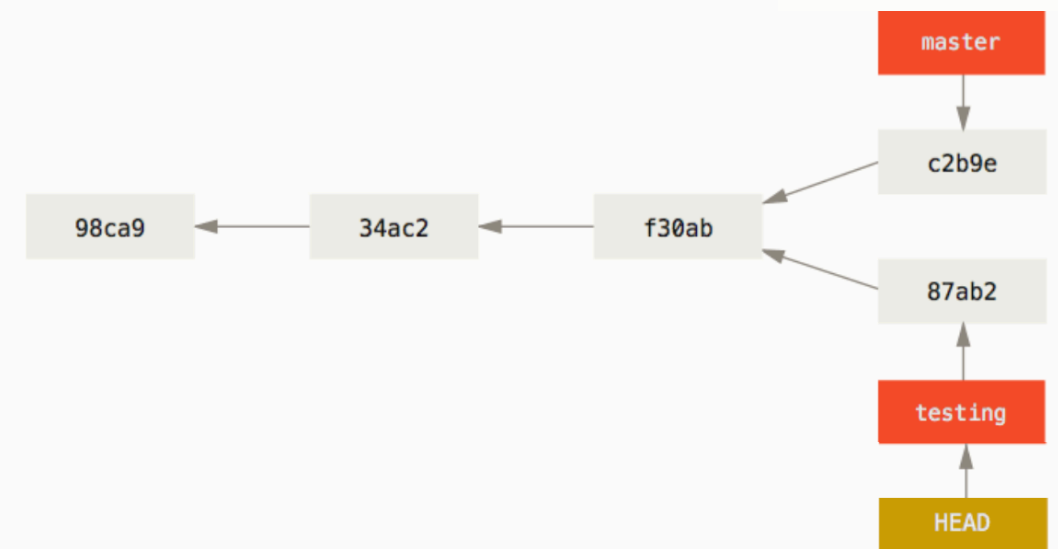


Figure 17. Divergent history

<https://git-scm.com/book/en/v2/Git-Branching-Branched-in-a-Nutshell>

```
$ git diff
```

## Complete Overview

**Display changes** to your tracked files

```
$ git diff
```

To be precise: Differences between working directory and staging area ➡ only unstaged changes

Helpful to inspect what you have done

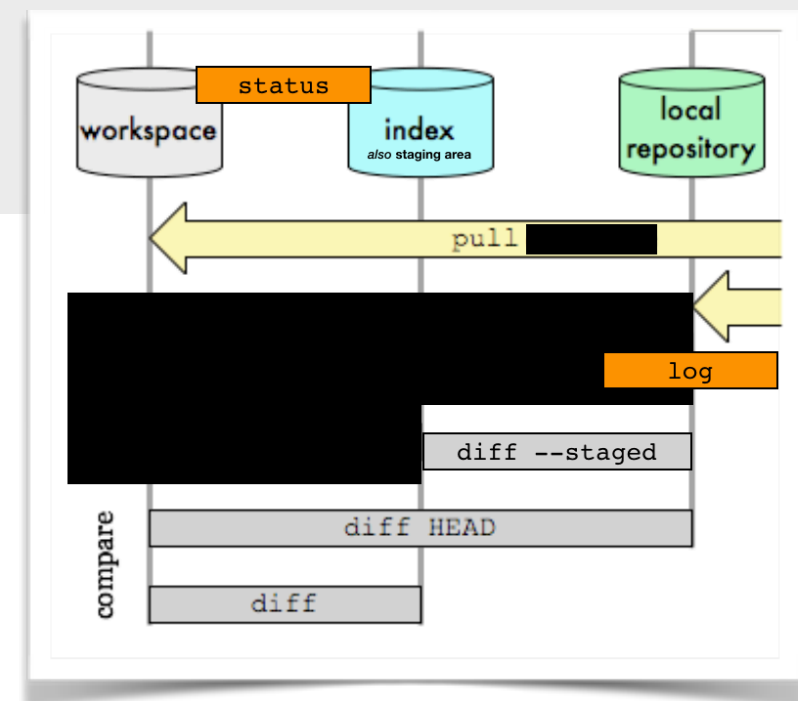
```
$ git diff HEAD
```

Differences between working directory and HEAD (last commit)

```
$ git diff --staged
```

Differences between staging area and HEAD

*already discussed*



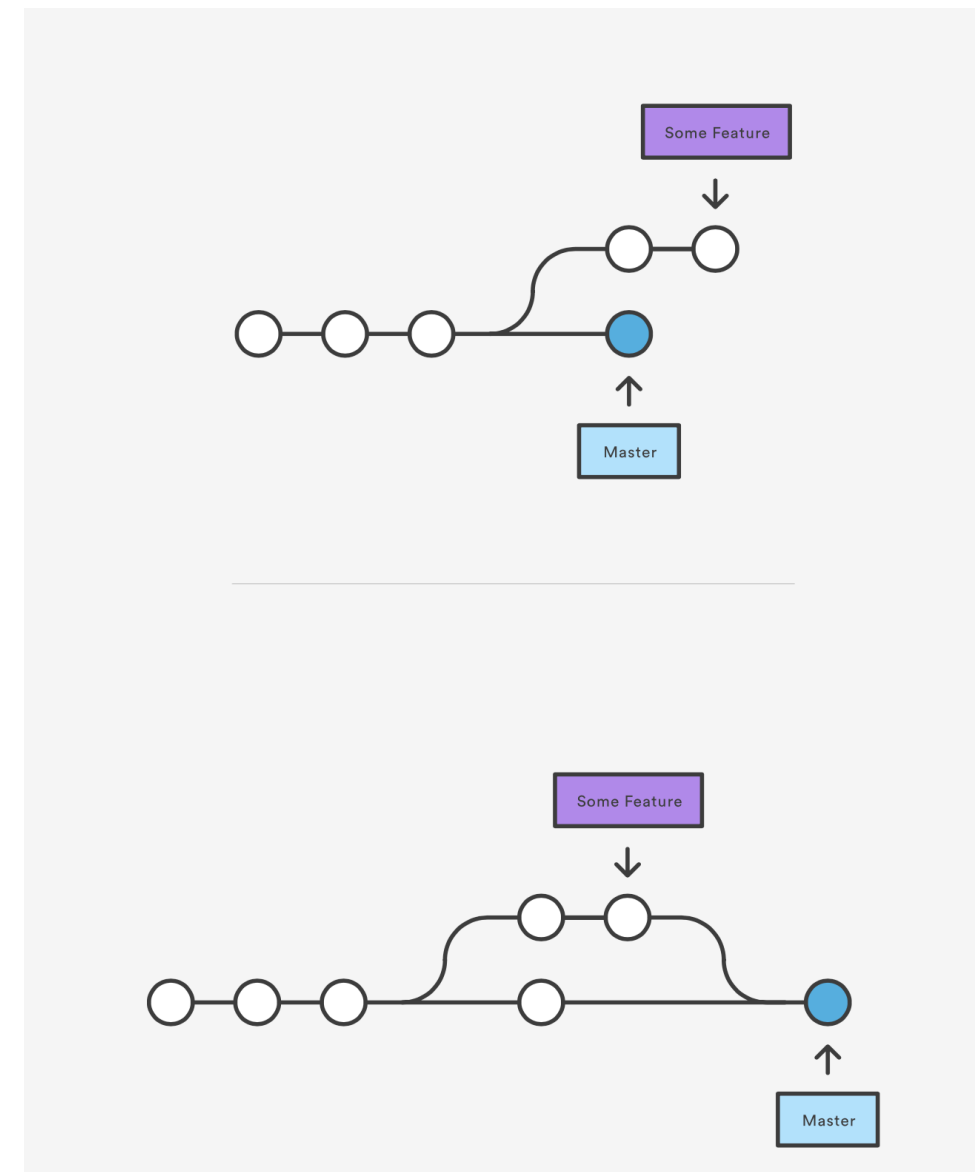
```
$ git merge
```

**Merge** changes in checked out branch

```
$ git merge <feature_branch>
```

Smart automatic three-way merges

Only changes the checked out branch (ensure you are on the correct branch)

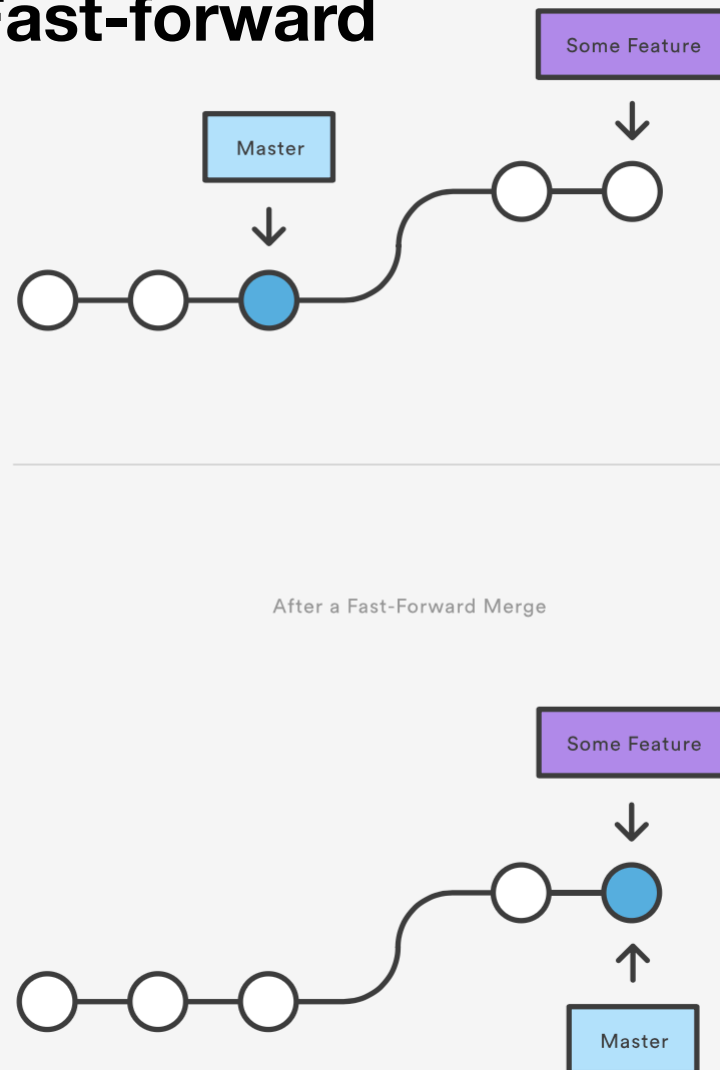


<https://www.atlassian.com/git/tutorials/using-branches/git-merge>

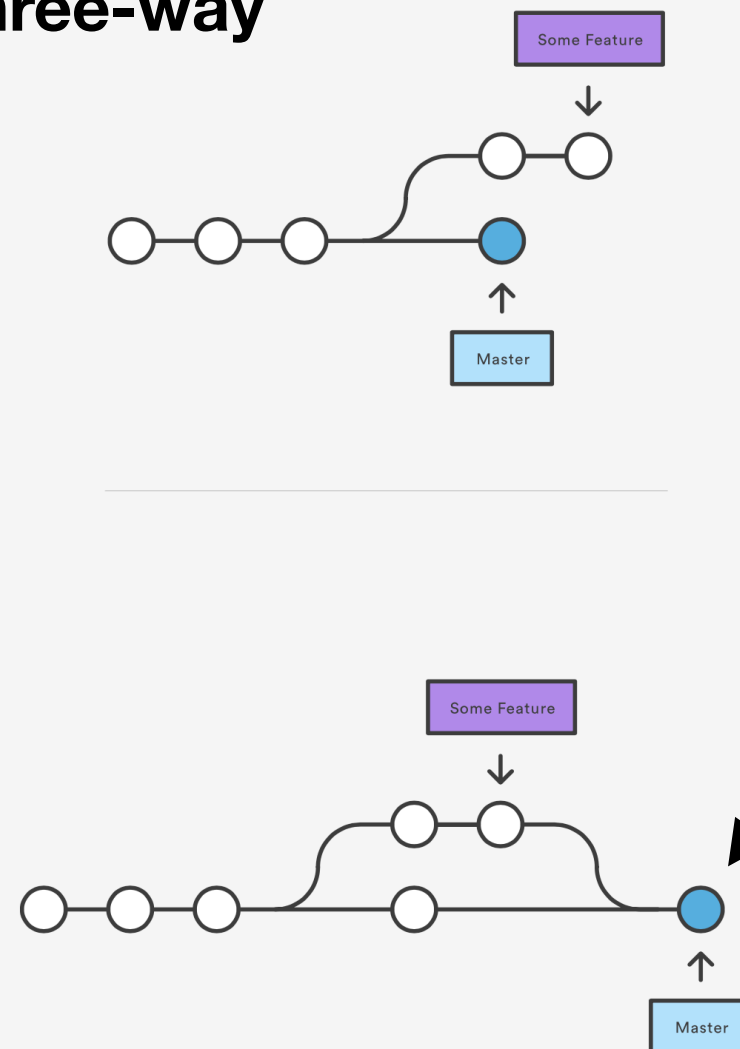
# \$ git merge

## Two different merges:

### Fast-forward



### Three-way



Only three-way merges have merge commits and potential merge conflicts

```
$ git merge
```

## Merge conflicts

```
$ git merge <branch_name>
Auto-merging <file>
CONFLICT (content): Merge conflict in <file>
Automatic merge failed; fix conflicts and then commit the result.
```

Conflicts if same part of file (*hunk*) is changed in both branches

## Resolve

1. Run `git status` to see „unmerged paths“
2. Find problematic hunks: Highlighted in files by  
`<<<<<<<`, `=====`, `>>>>>>>`
3. Create the intended code version and remove `<<<<<<<`, ...
4. Then `git add <file_with_merge_conflict>`
5. `git commit` (Auto-generated merge commit message)

```
...
<<<<<<< HEAD
std::cout << "Hello!";
=====
std::cout << „Goodbye!";
>>>>>>> say_goodbye_branch
...
```

```
...
std::cout << „Goodbye!";
...
```

# Commands

config

init

status

add

commit

diff

log

branch

checkout

merge

**clone**

**push**

**pull**



# Exercise 2

config  
init  
status  
add  
commit  
diff  
log

**branch  
checkout  
merge**

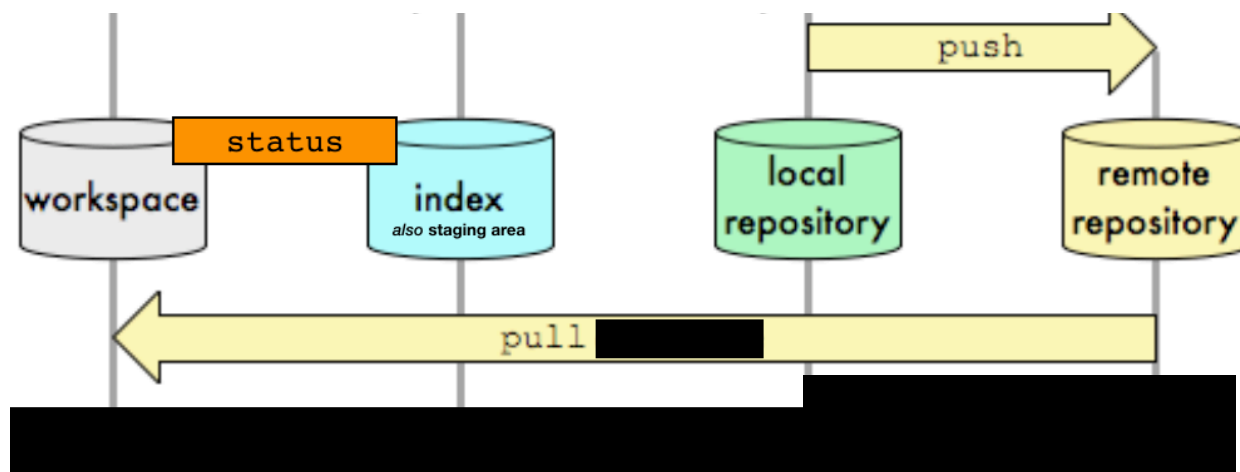
**Solutions**  
can be found at the  
end of the slides

# Exercise 2

1. Create a new branches and check that you created them by looking at the branch list.
2. Delete the new branches right away without risking data loss.
3. Create and directly switch to another new branch (e.g. name it add\_readme) using only one command. Check again that you created the branch and that it is checked out.
4. Add a new file on the branch (e.g. a README) and commit the new file.
5. Switch back to master. Verify that the new file is gone. Merge the branch with the new file (Notice that this was a fast-forward merge). Check the history and that the new file is now reappeared. Safely delete the merged branch, which is possible now that the changes are in master.
6. Provoke a merge conflict by creating a new branch (e.g. name it edit\_sample\_file) and change the sample file from Exercise 1. Commit the changes on the new branch. Edit the same part of the file back on master and also commit the changes. Now, try merging the new branch with the edited sample file.
7. Fix the merge conflict.

# Interacting with Remotes

- So far everything were local operations
- Following interactions with a remote repository require network connection
- Remote repositories enable **collaboration** and backup
- Local repository has to be **manually synced** with remote repository



## Advanced:

Note: In the following only tracking branches are used to interact with a remote repository to keep it simple.

```
$ git clone
```

**Clone (download) a remote repository**

```
$ git clone <link_to_repository>
```

Creates directory with project name in current directory

Remote repository is (by default) referred to as `origin`

You can also clone on the same machine locally

**See all remote branches** of repository

```
$ git branch -r
```

Branches that are in remote are prefixed by `origin/` then the `<branch_name>`

```
$ git pull
```

**Checkout** a branch of the repository (as usual)

```
$ git checkout <branch_name>
```

Do not use the `origin/` prefix here

**Update** a branch with the new version from the remote repository

```
$ git pull
```

... while the branch is checked out

Changes your working directory

**Make sure you pull before committing and merging to stay in sync!** (especially on master, maybe someone else updated it)

```
$ git push
```

**Create** a new branch in the remote repository

```
$ git push -u origin <branch_name>
```

from the currently checked out branch

**Update the remote branch** from the local branch afterwards

```
$ git push
```

Only changes that are committed are pushed

If the remote and local history diverge (e.g. forgot to pull before committing) pushes will be rejected

**Make sure you push after committing and merging to stay in sync!**

# The Rest

## Other useful commands worth looking up yourself

```
$ git stash
```

Quickly stash away your changes for later to obtain clean working tree

```
$ git revert  
$ git reset
```

Undoing changes and commits  
—> good tutorial under this [\[Link\]](#)

```
$ git blame  
    <file>
```

See line for line, who and which commit is responsible for the last change to this line

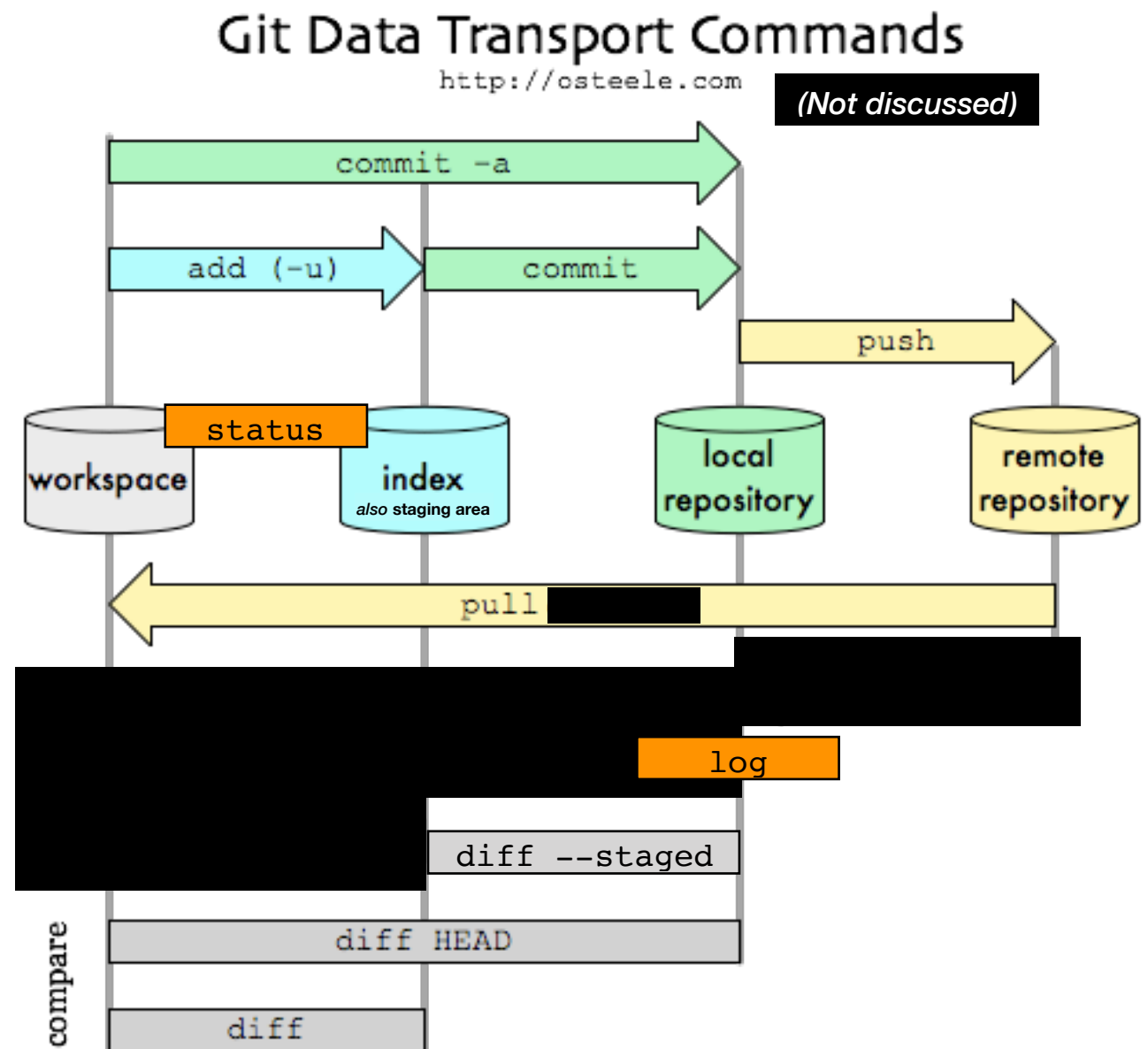
```
$ git bisect
```

Bisect the git history to find which commit introduced a bug

# Summary

**config**  
**init**  
**status**  
**add**  
**commit**  
**diff**  
**log**

**branch**  
**checkout**  
**merge**  
**clone**  
**push**  
**pull**





# Good Resources

## Beginner:

- [Bitbucket Tutorials for git](#)
- [Resources to learn by Github \(includes interactive tutorials\)](#)
- [Git Documentation](#)

## More advanced:

- [Pro Git by Scott Chacon and Ben Straub: THE git book \(free\)](#)
- [Good talk: Introduction to git with Scott Chacon of GitHub](#)
- [Good talk: Linus Torvalds \(creator of git\) on git](#)

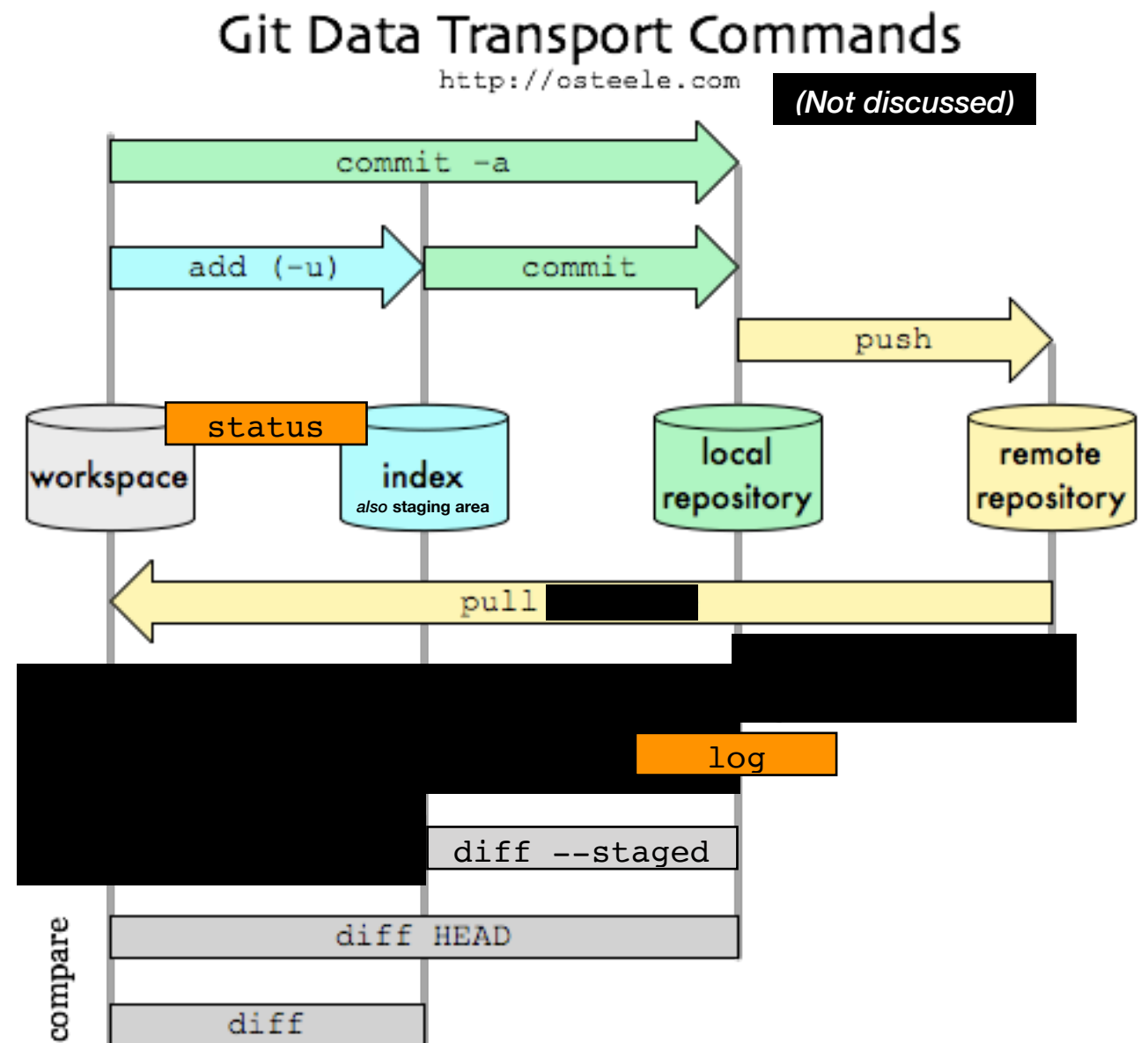
# Summary

Feedback and questions  
welcome!

[staudenmaier@fias.uni-frankfurt.de](mailto:staudenmaier@fias.uni-frankfurt.de)

**config**  
**init**  
**status**  
**add**  
**commit**  
**diff**  
**log**

**branch**  
**checkout**  
**merge**  
**clone**  
**push**  
**pull**



*Online Slides with the hands-on exercises:*

[CRC Redmine Z02 Project Wiki](#) or [Transport Meeting Website](#)

# Solutions to Exercise 1

```
# Part 1 #
$ git config --global user.name <your_name>
$ git config --global user.email <your_email>
$ less ~/.gitconfig

# Part 2 #
$ mkdir sample_project; cd sample_project
$ git init
$ ls -a # see hidden .git directory

# Part 3 #
$ vi sample_file.cpp # create sample file
$ git status
$ git add sample_file.cpp
$ git status
$ git commit # editor open, type commit message, save and quit

# Part 4 #
$ vi sample_file.cpp # modify sample file
$ git diff
$ git status
$ git add sample_file.cpp
$ git status
$ git commit # editor open, type commit message and save

# Part 5 #
$ git log
```

# Solutions to Exercise 2

## Part 1/2

```
# Part 1 #
$ git branch new_branch
$ git branch

# Part 2 #
$ git branch -d new_branch # small d important

# Part 3 #
$ git checkout -b add_readme
$ git branch # star marks the currently checked out branch

# Part 4 #
$ vi README.md # create README file
$ git status
$ git add README.md
$ git commit
$ git status # shows working tree clean, so we can check out another branch

# Part 5 #
$ git checkout master
$ ls # README is gone again
$ git merge add_readme # notice it says fast-forward merge
$ git log # has commit from add_readme
$ ls # README file now on master
$ git branch -d add_readme
```

**Builds on  
repository  
created for  
Ex. 1**

# Solutions to Exercise 2

## Part 2/2

```
# Part 6 #
$ git checkout -b edit_sample_file
$ vi sample_file.cpp # modify file
$ git diff # always check you changes
$ git commit -a # only small change
$ git checkout master
$ vi sample_file.cpp # modify same part/line of file
$ git diff
$ git commit -a
$ git merge edit_sample_file # should have a merge conflict

# Part7 #
$ git status # see unmerged paths
$ vi sample_file.cpp # create intended code version + rm comments
$ git add sample_file
$ git commit # auto-generated merge commit message
```