GIT Advanced

Anthony Baire

Université de Rennes 1 / UMR IRISA

June 1, 2022

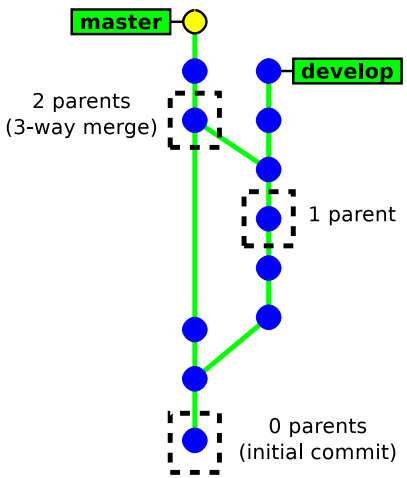
Summary

1. Reminders
2. Git's Guts
3. Specifying revisions
4. Playing with your index
5. Rewriting your history
6. Interoperating with other version control systems
7. Submodules and Subtrees
8. Managing a collection of patches
9. Managing large files
10. Scaling

Part 1.
Reminders

How GIT handles its history

- Git's history is a Direct Acyclic Graph
- A commit may have 0, 1 or more parents
- A commit is identified by the sha1sum of its content and recursively of all its ancestors
- Their is no formal branch history.
  A branch is just a reference to the last commit



The staging area (aka the "index")

Usual version control systems provide two spaces:

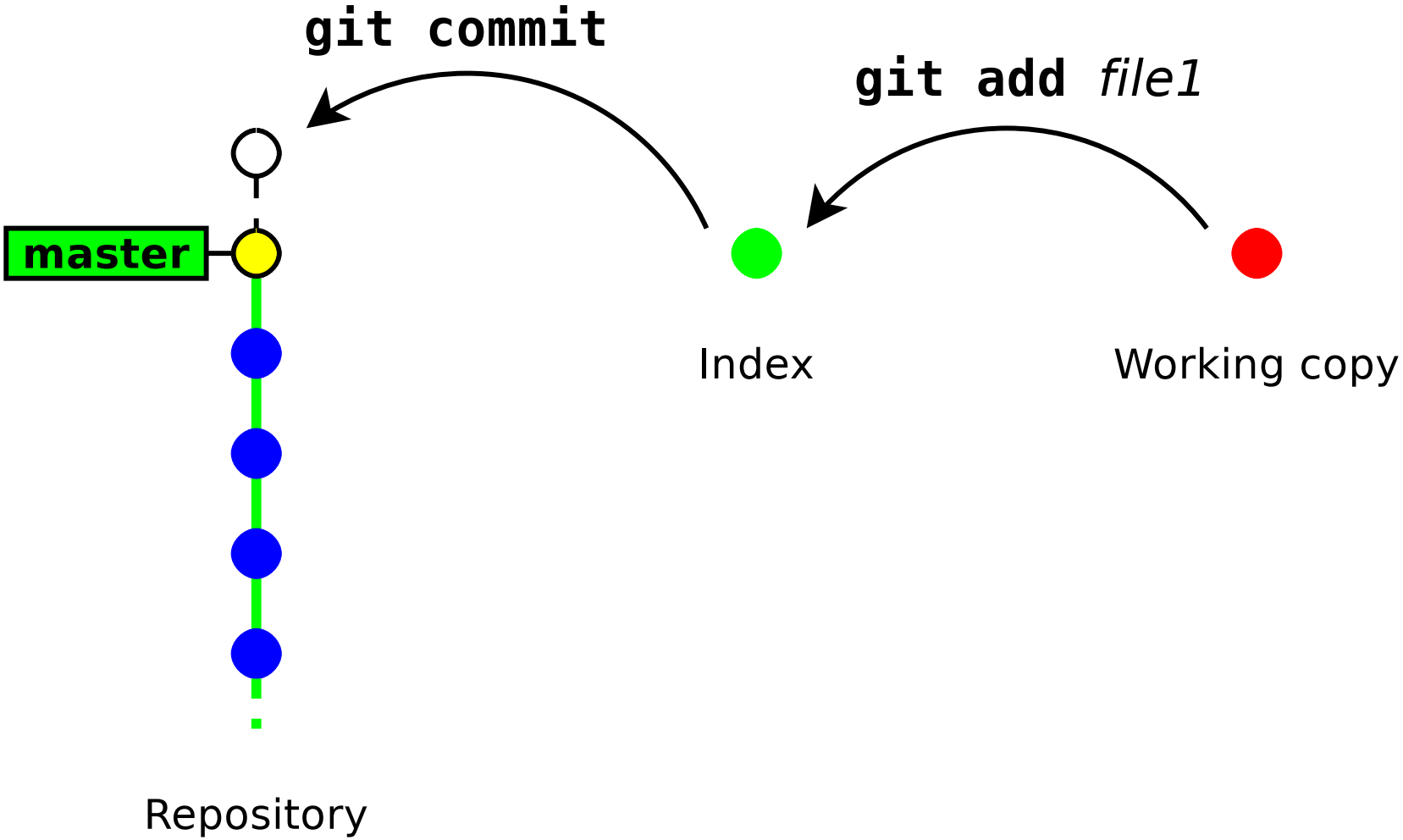
- the **repository**
  (the whole history of your project)
- the **working tree** (or **local copy**)
  (the files you are editing and that will be in the next commit)

Git introduces an intermediate space : the **staging area** (also called **index**)

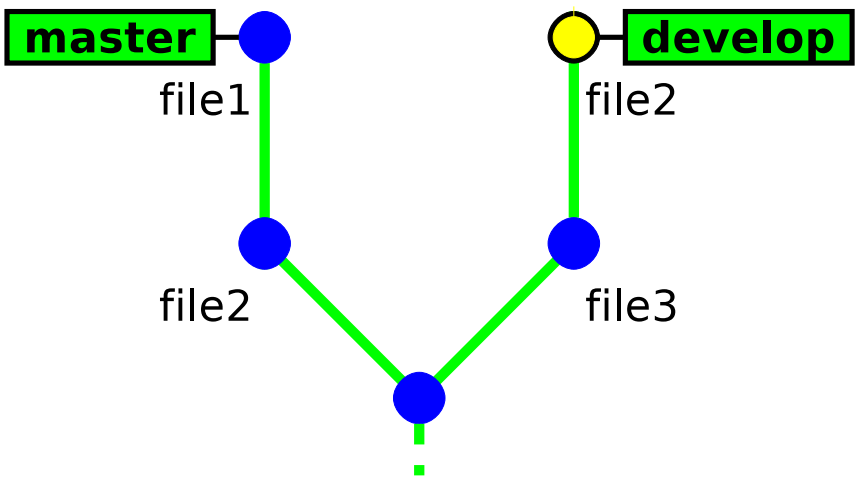
The index stores the files scheduled for the next commit:

- `git add` *files* $\rightarrow$ copy files into the index
- `git commit` $\rightarrow$ commits the content of the index

The staging area (aka the "index")

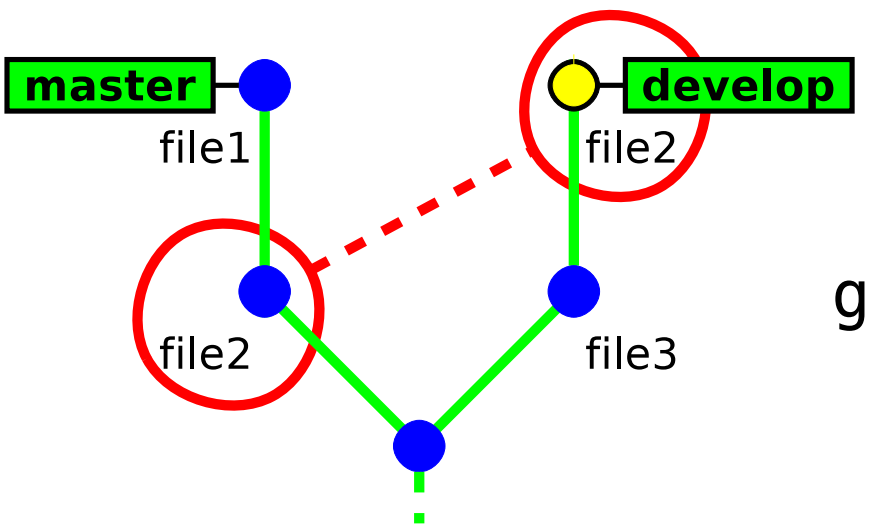


`git merge` and the index



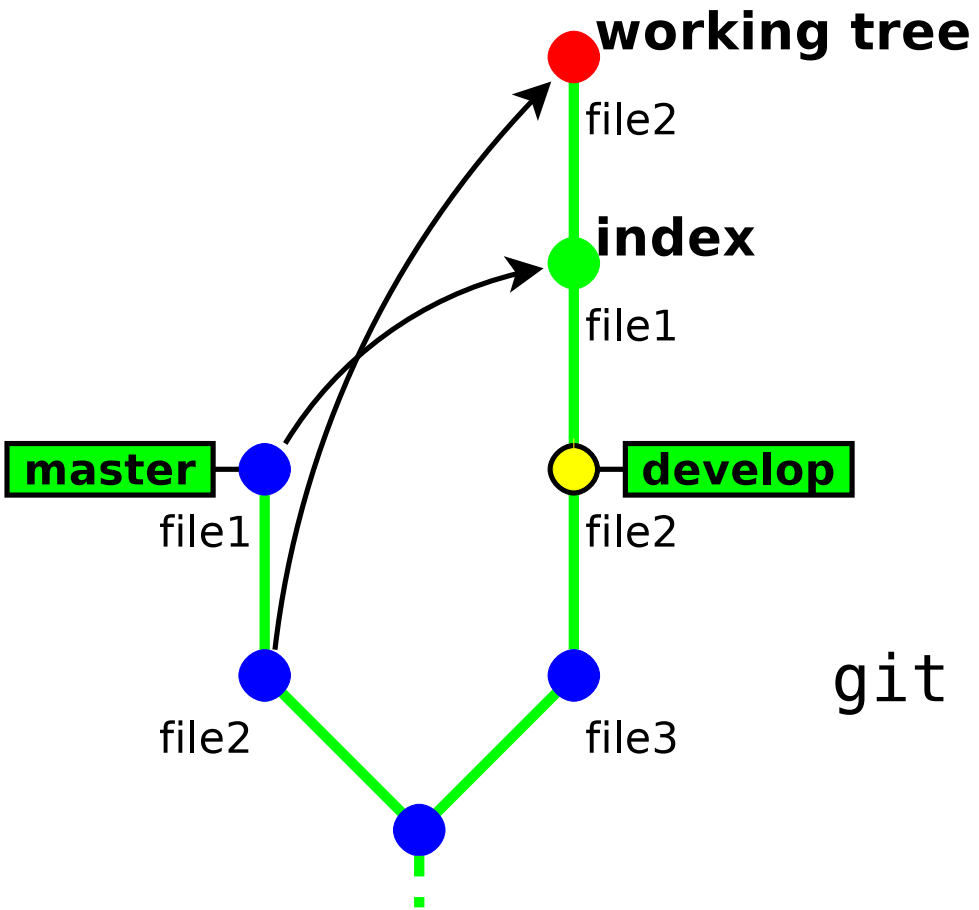
`git merge` and the index

!! conflict !!



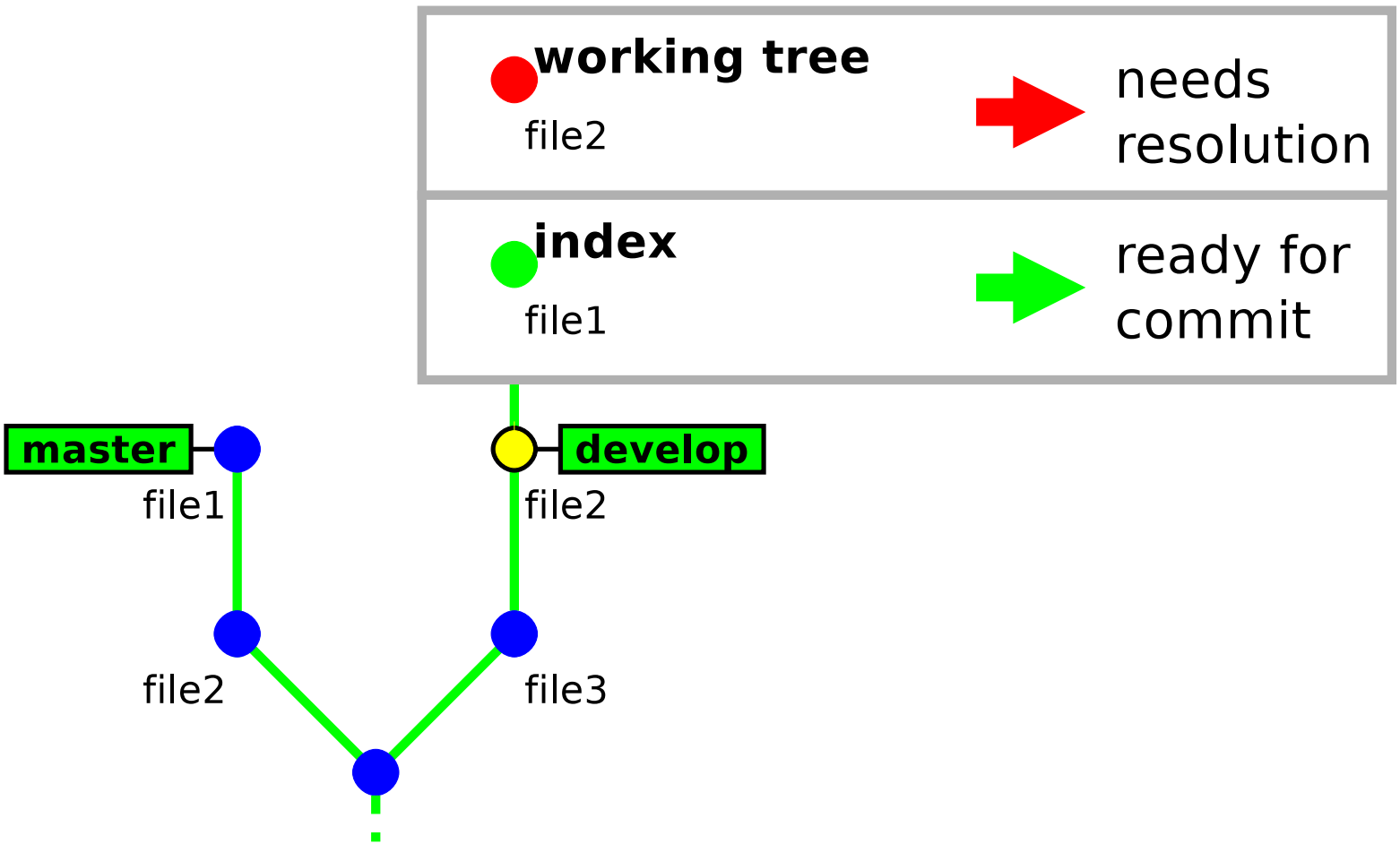

```
git merge master
```

git merge and the index

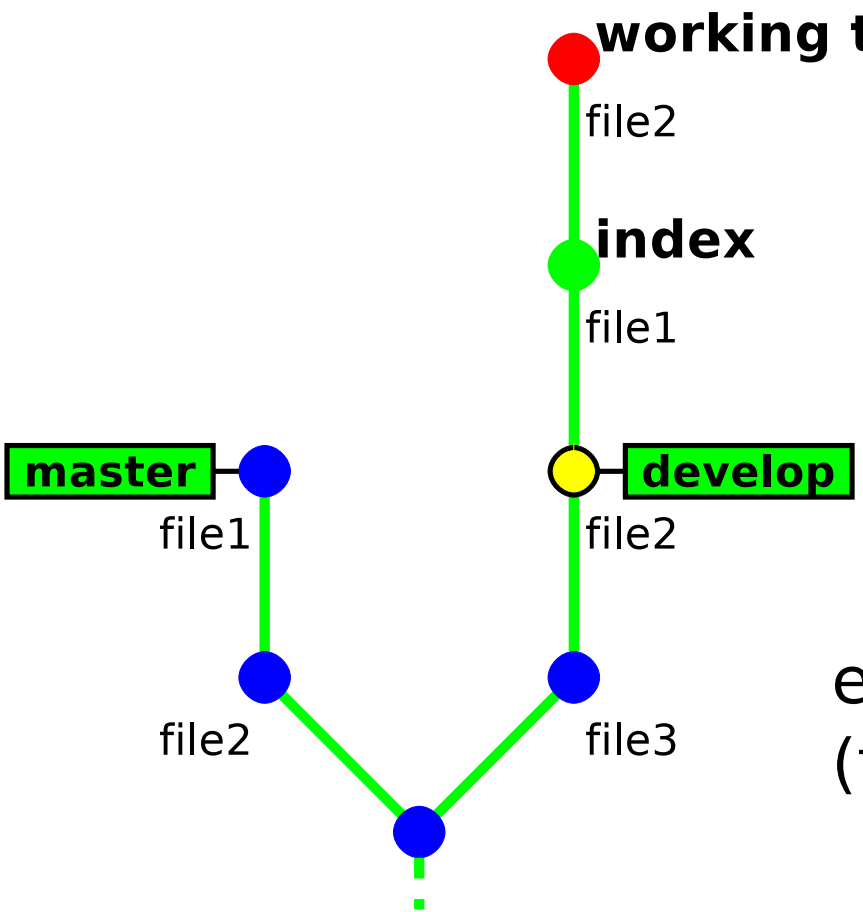



```
git merge master
```

`git merge` and the index

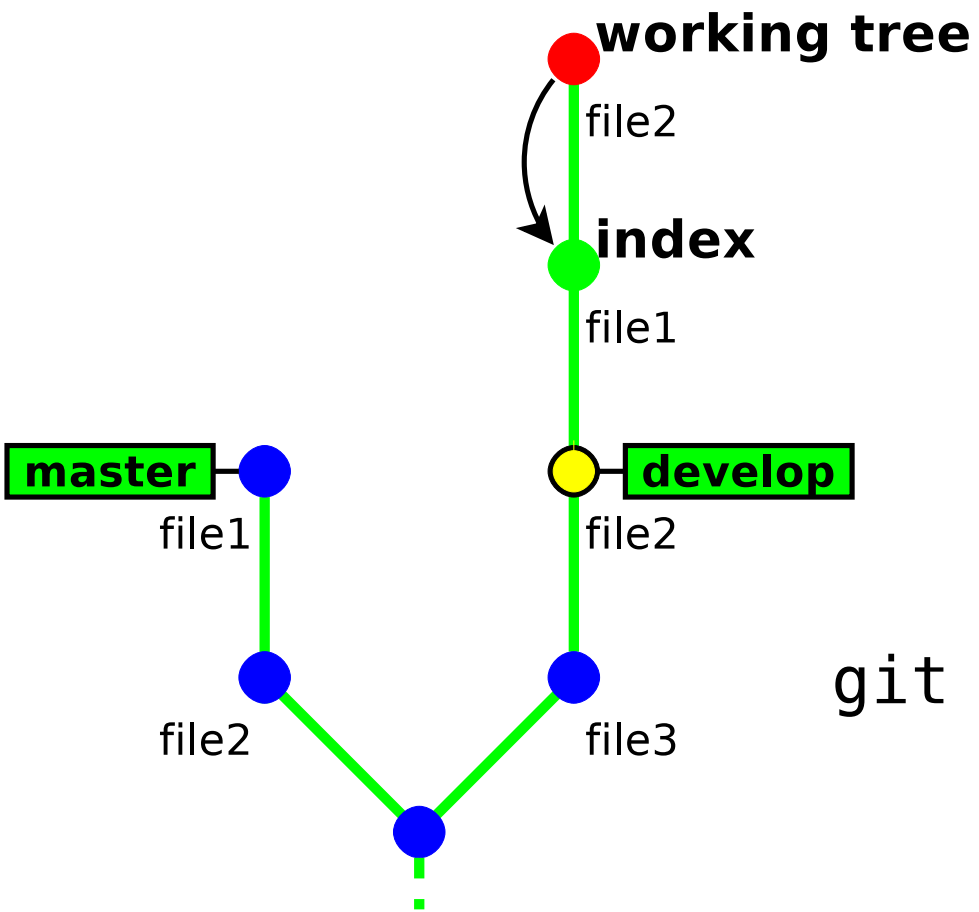


`git merge` and the index



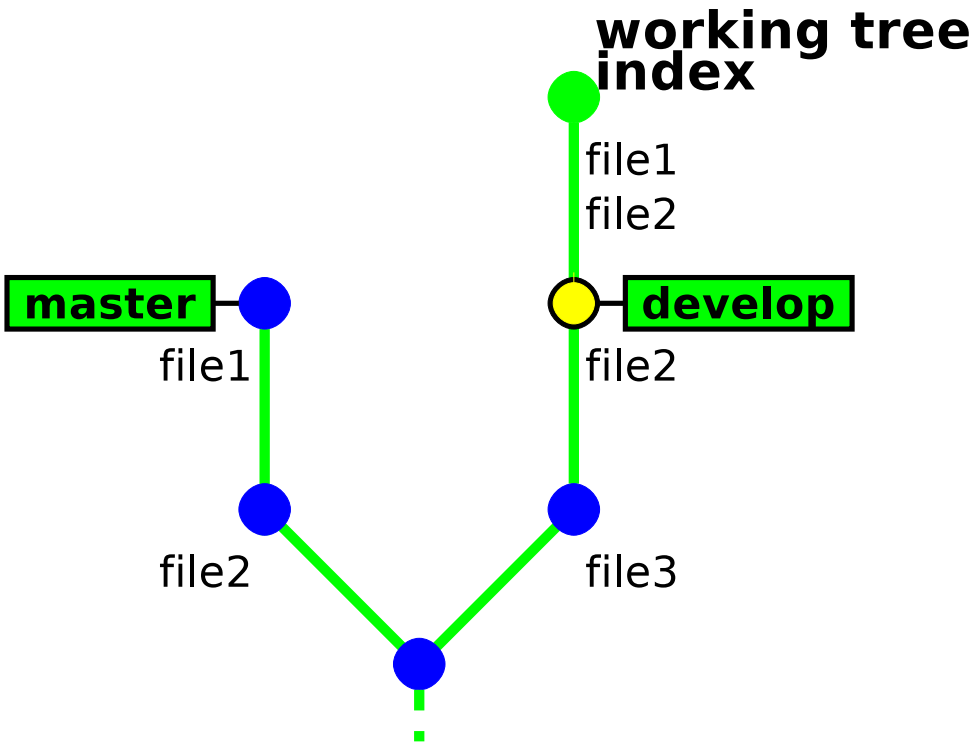
edit file2
(to resolve conflicts)

`git merge` and the index

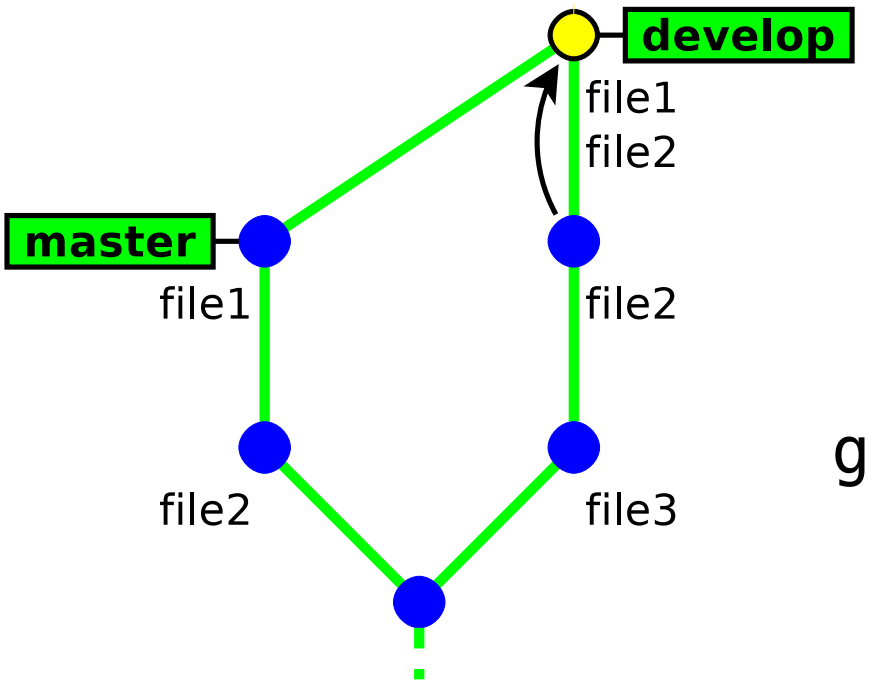


`git add file2`

`git merge` and the index



`git merge` and the index



`git commit`

Part 2.
Git's Guts

Layout of a git repository

```
$ ls .git
branches/  description  hooks/  info/  objects/  refs/
config     FETCH_HEAD   HEAD    index  logs/     ORIG_HEAD
```

Configuration files		
	`config`	main config file
	`description`	description of the repository
	`hooks/`	hook scripts
Database (immutable objects)		
	`objects/`	commits, trees, blobs, tags
State files (mutable objects)		
References (branches & tags)		
	`refs/`	raw store (one file per ref)
	`packed-refs`	packed store (all refs in the same file)
	`branches/`	remote references (deprecated[1])
Special references		
	`HEAD FETCH_HEAD ORIG_HEAD MERGE_HEAD CHERRY_PICK_HEAD`	
Index		
	`index`	state of the index
Logs	`logs/`	history of branches
Misc	`info/`	config files (`exclude attributes`) and cached state (`refs`)

[1] they are now stored as `refs/remotes/*`

Git Database

Git'd database contains four types of objects :

- **blob** (content of a file)
- **tree** (content of a directory)
- **commit**
- **tag** (tag annotations & GPG signatures)

Objects are stored in a hash table. Their key is the sha1sum of their content.

```
$ echo 'Hello World!' | git hash-object --stdin -t blob -w
980a0d5f19a64b4b30a87d4206aade58726b60e3

$ git cat-file blob 980a0d5f19a64b4b30a87d4206aade58726b60e3
Hello World!
```

The database is stored in `.git/objects/`

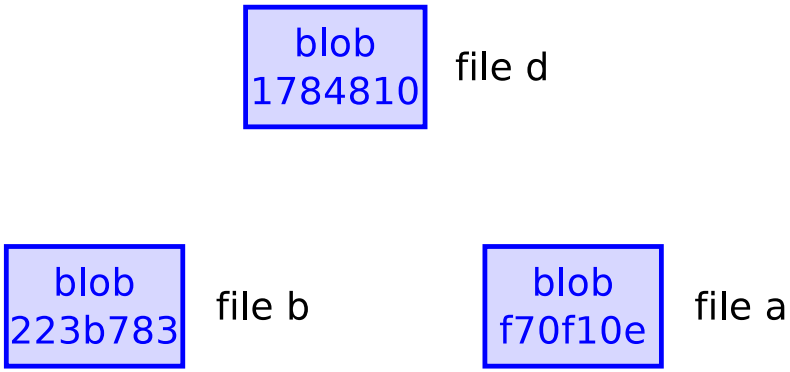
Git DB Example

commit #1

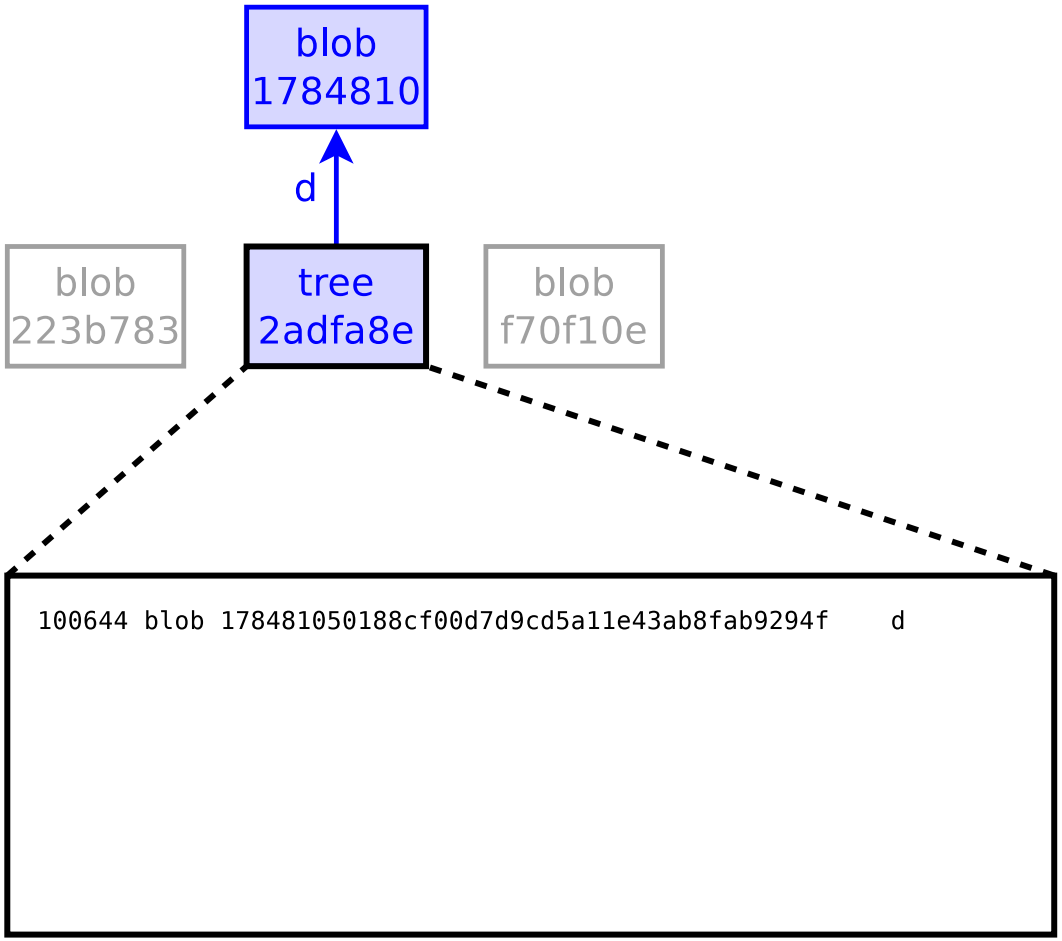
Initial commit with three files and sub-directory:

created file → a
created file → b
created file → c/d

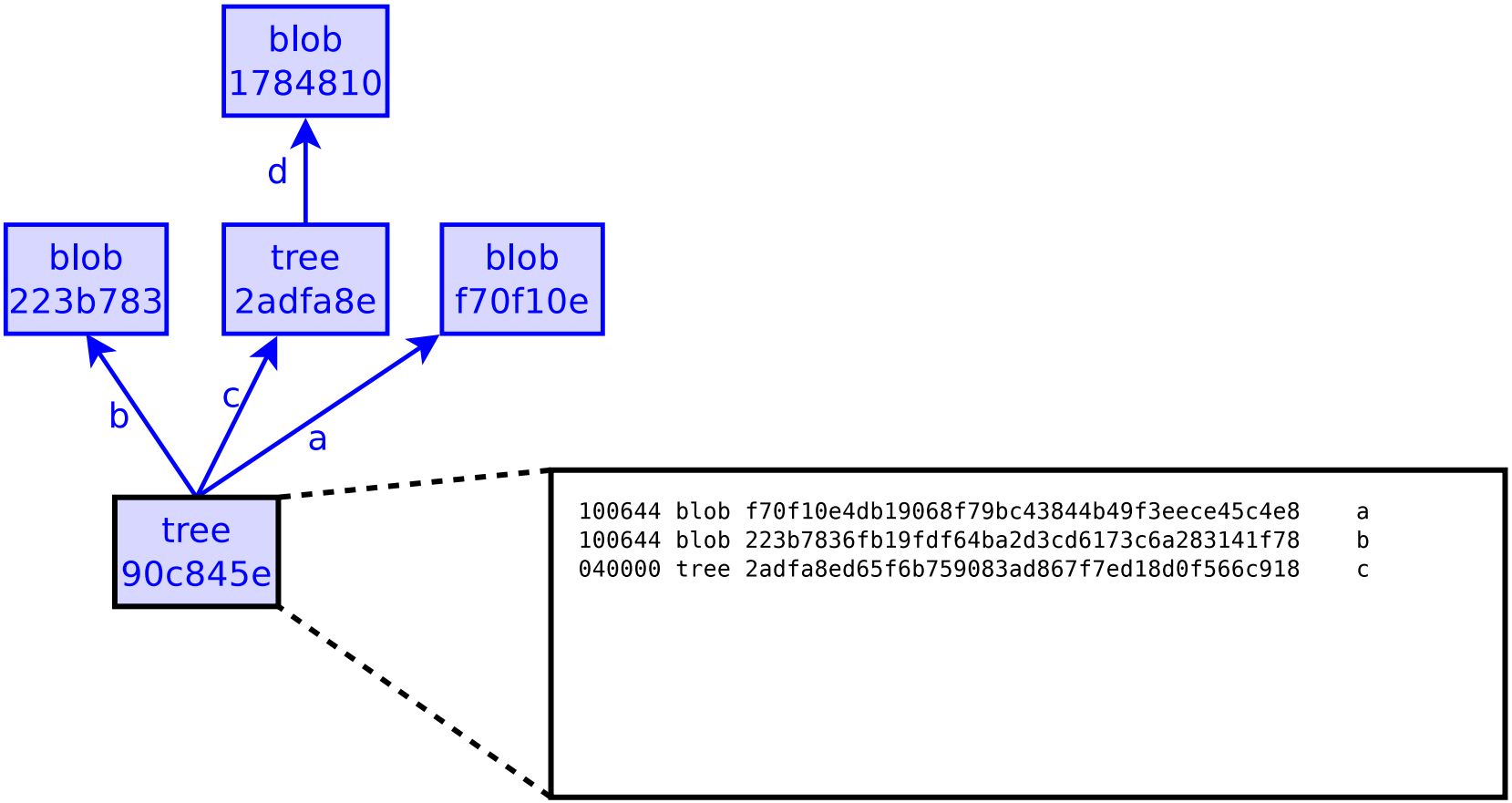
Git DB Example



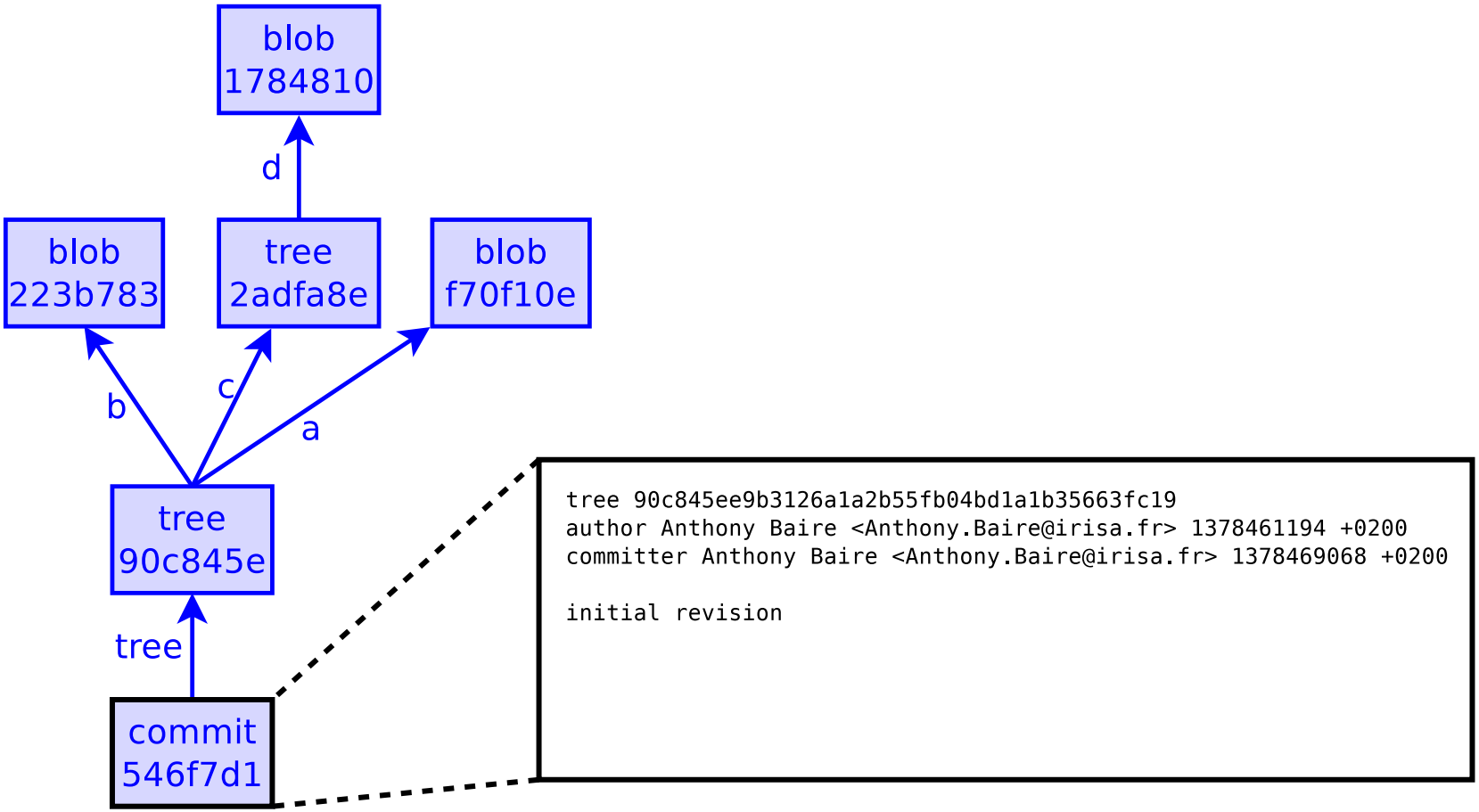
Git DB Example



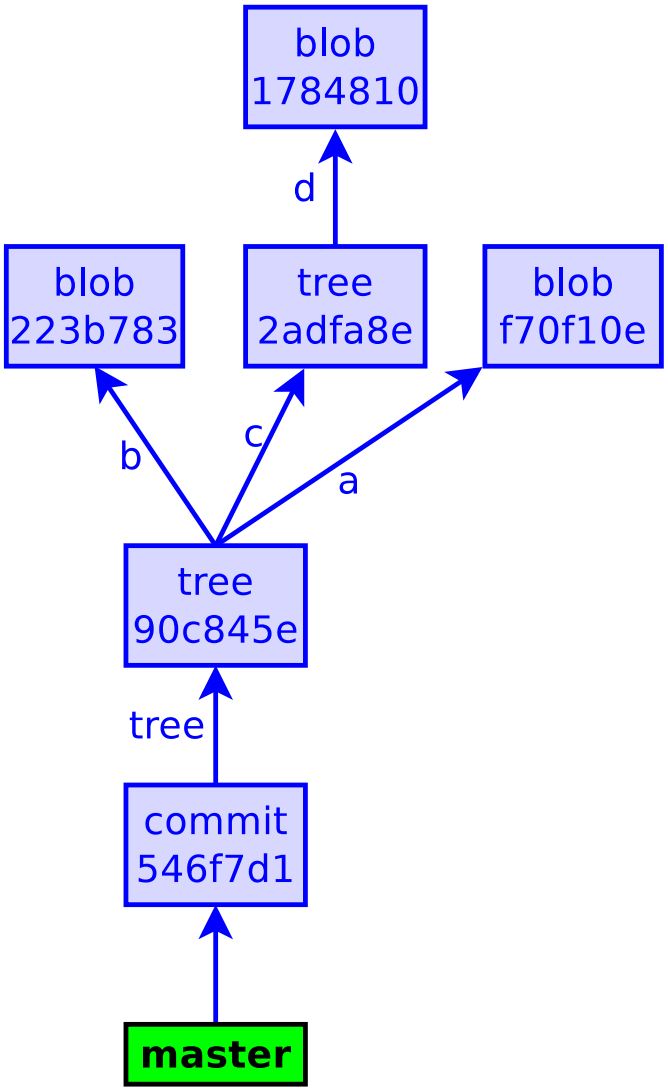
Git DB Example



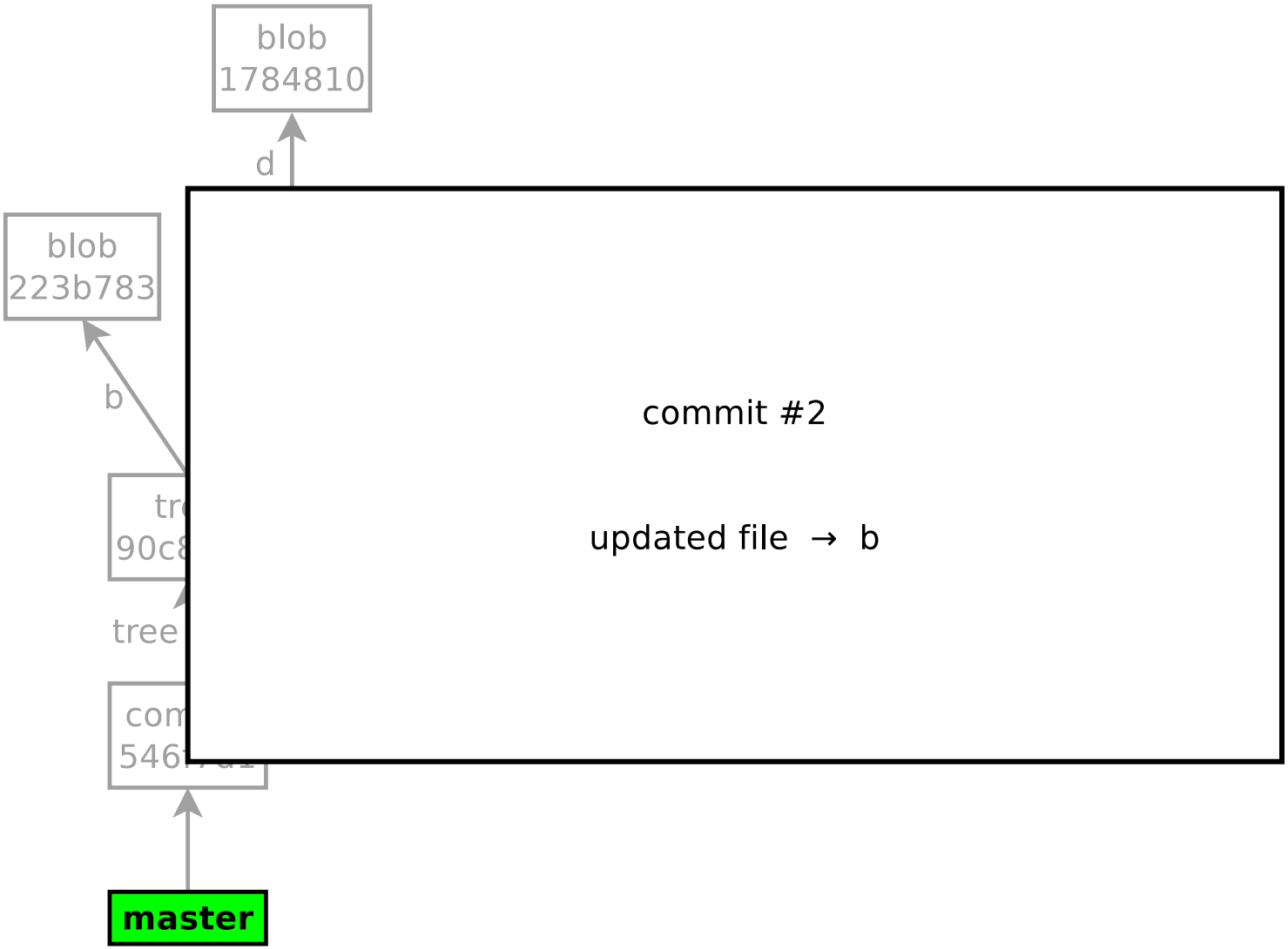
Git DB Example



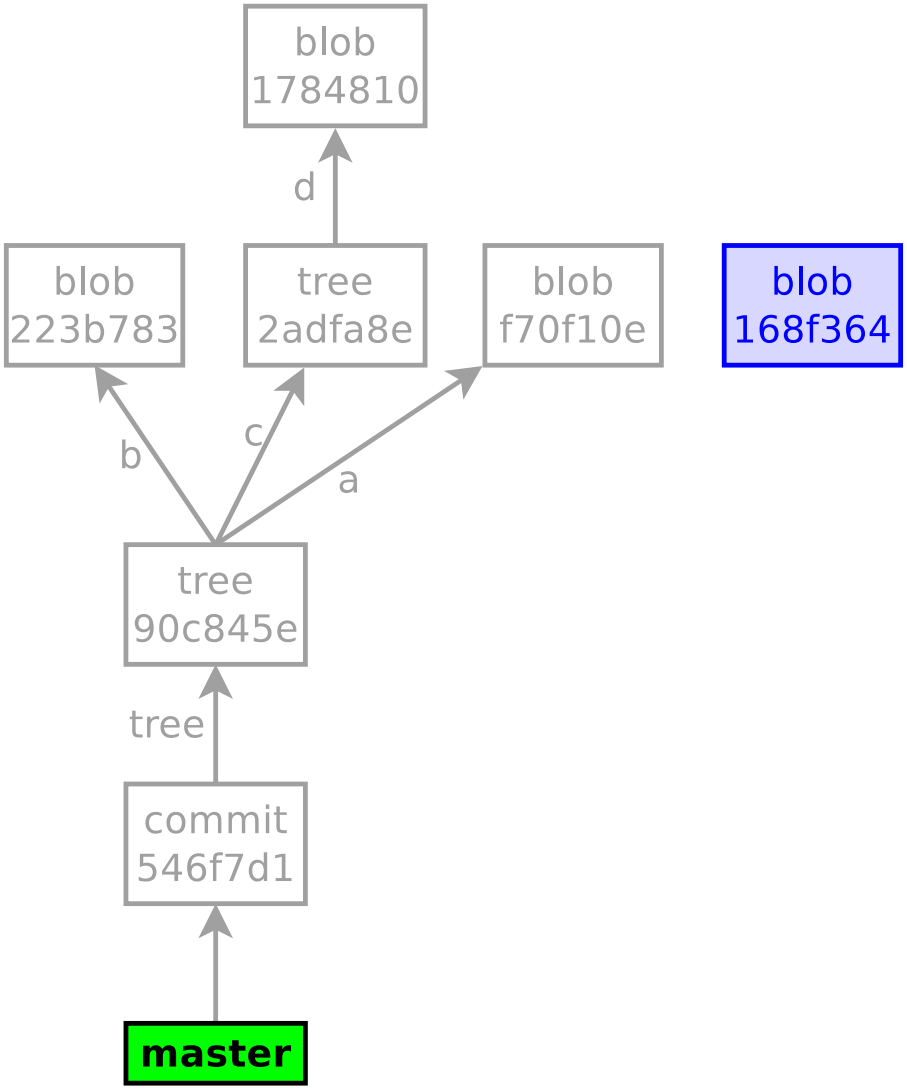
Git DB Example



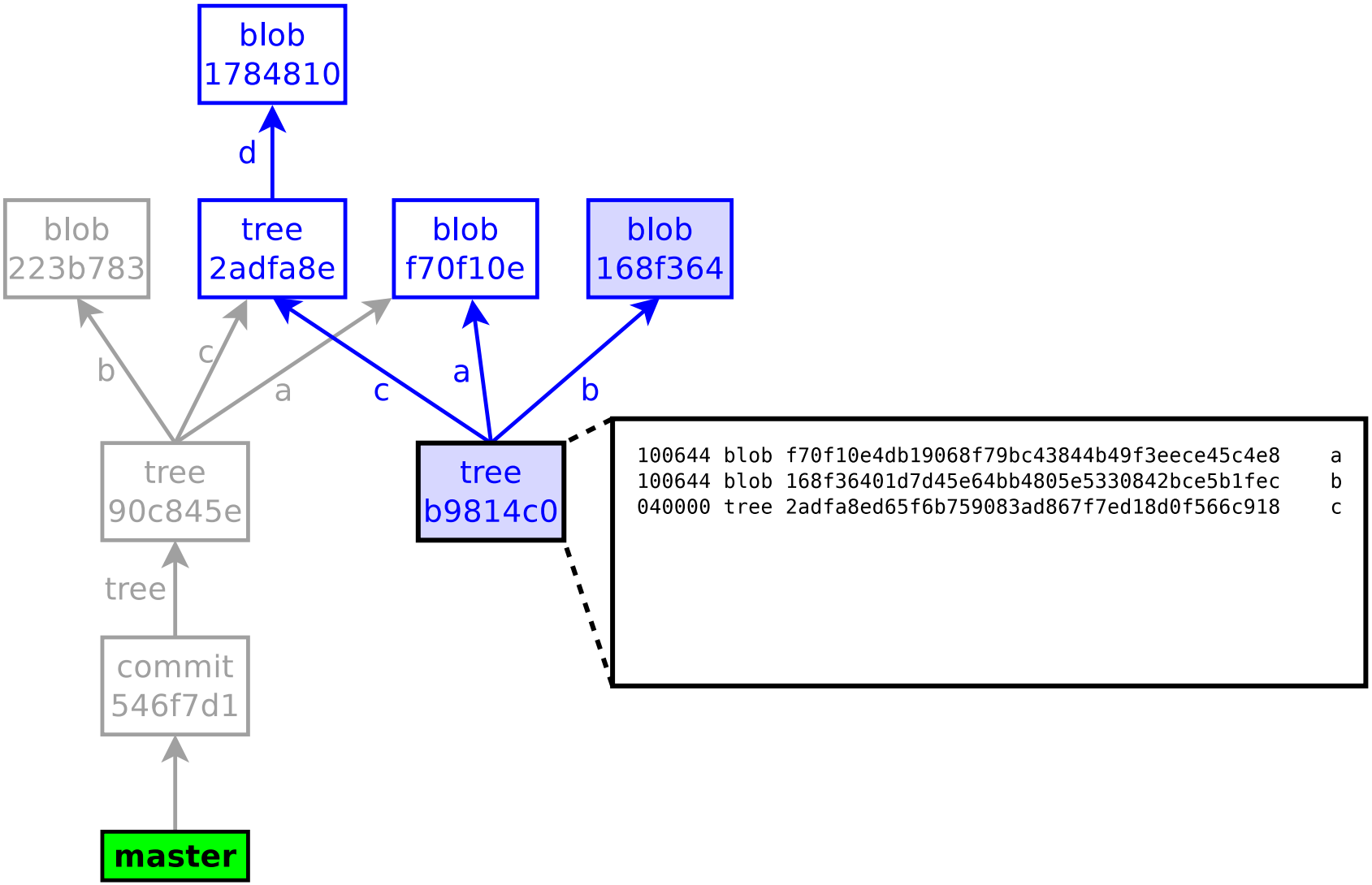
Git DB Example



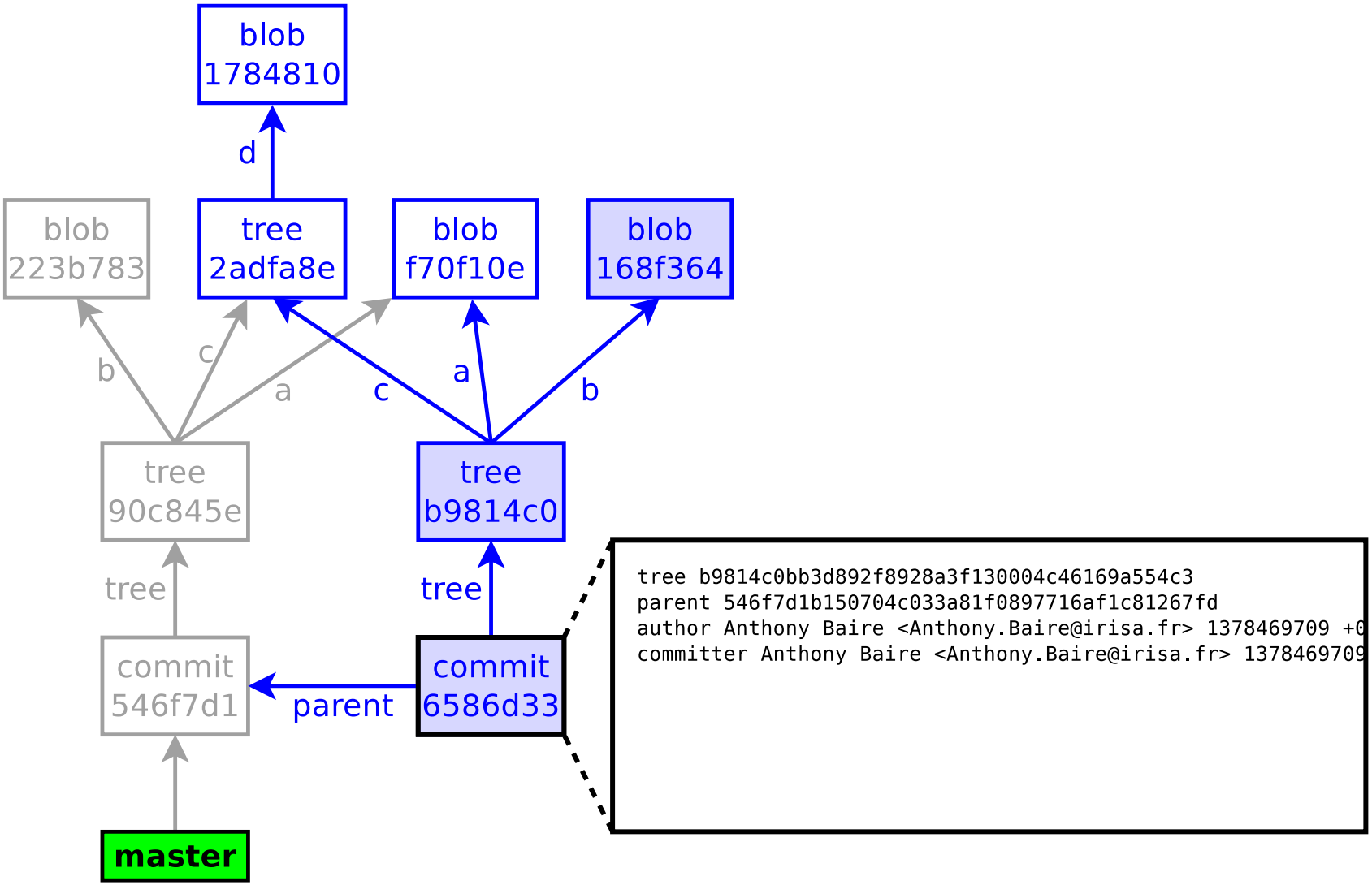
Git DB Example



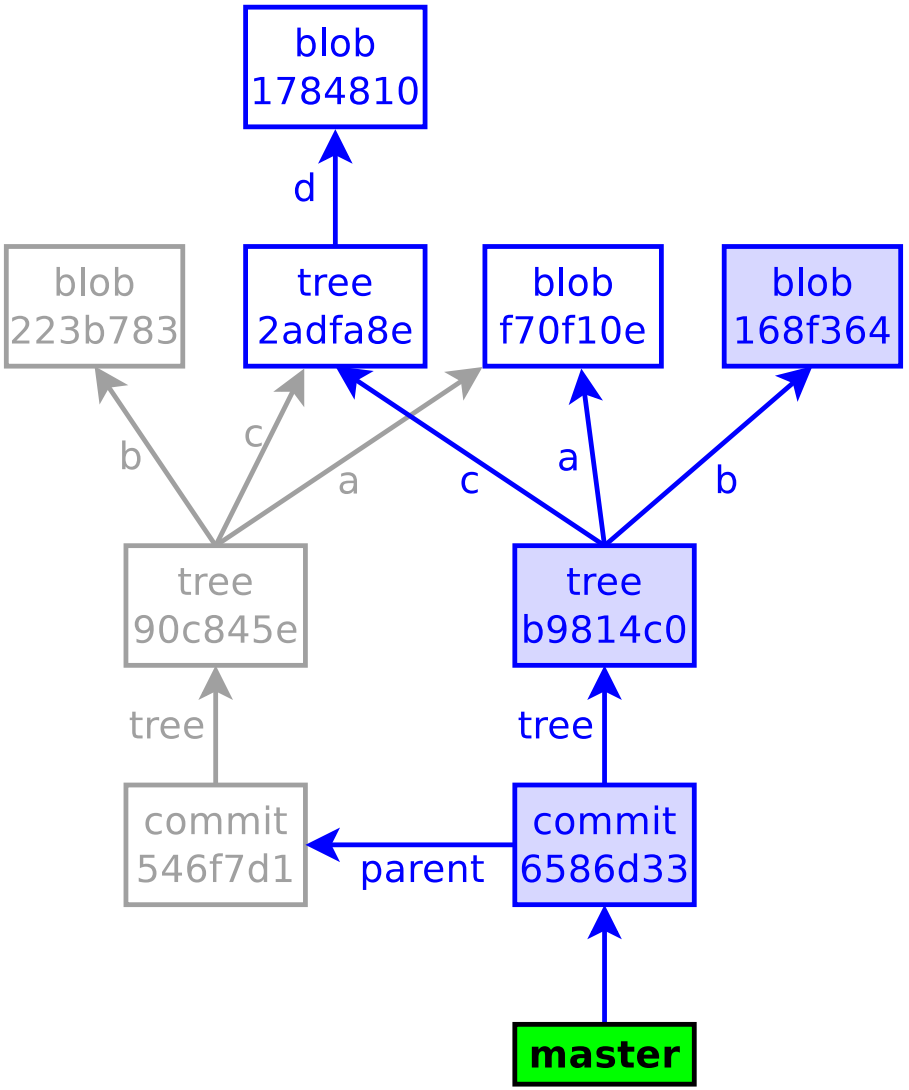
Git DB Example



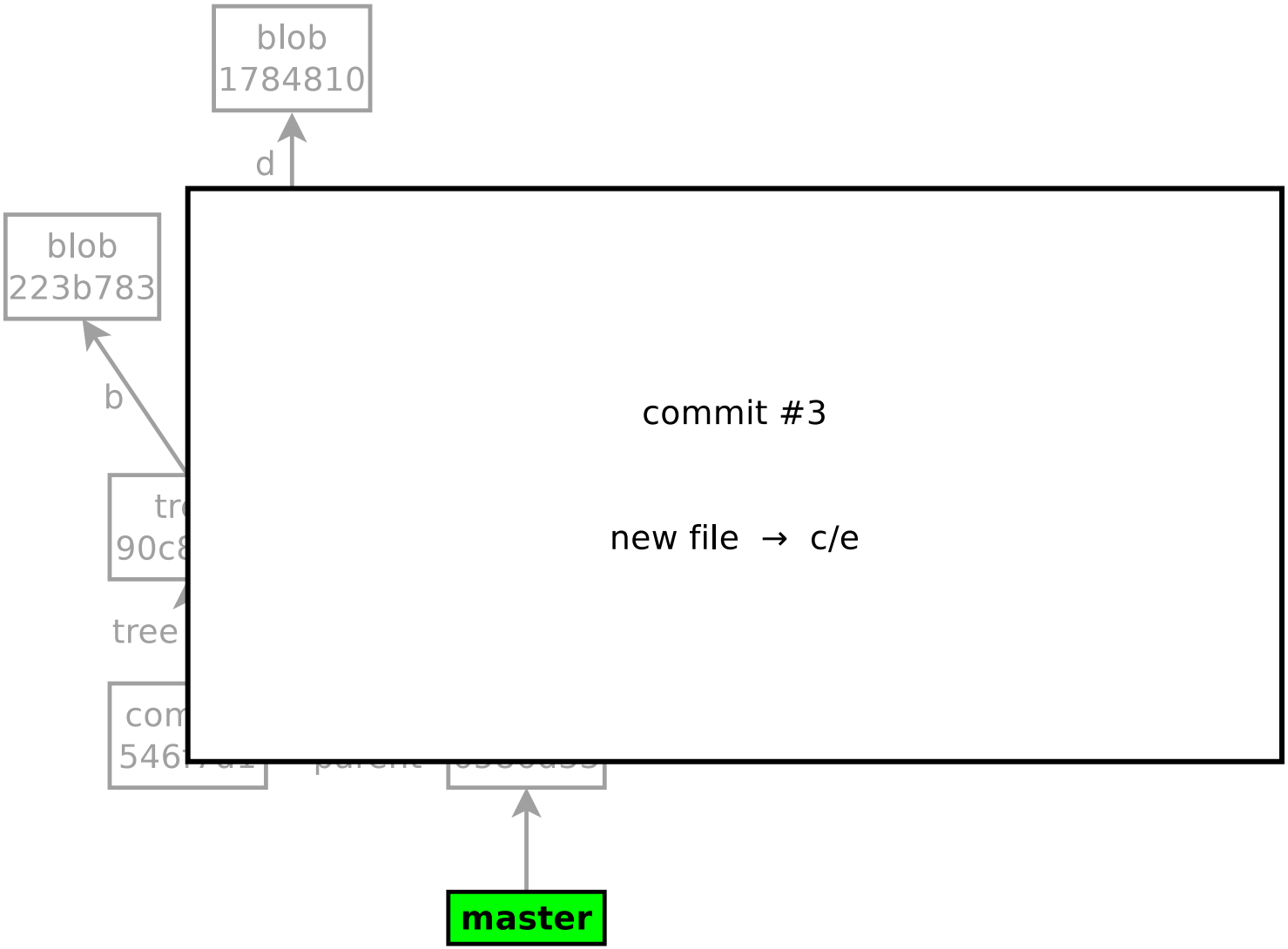
Git DB Example



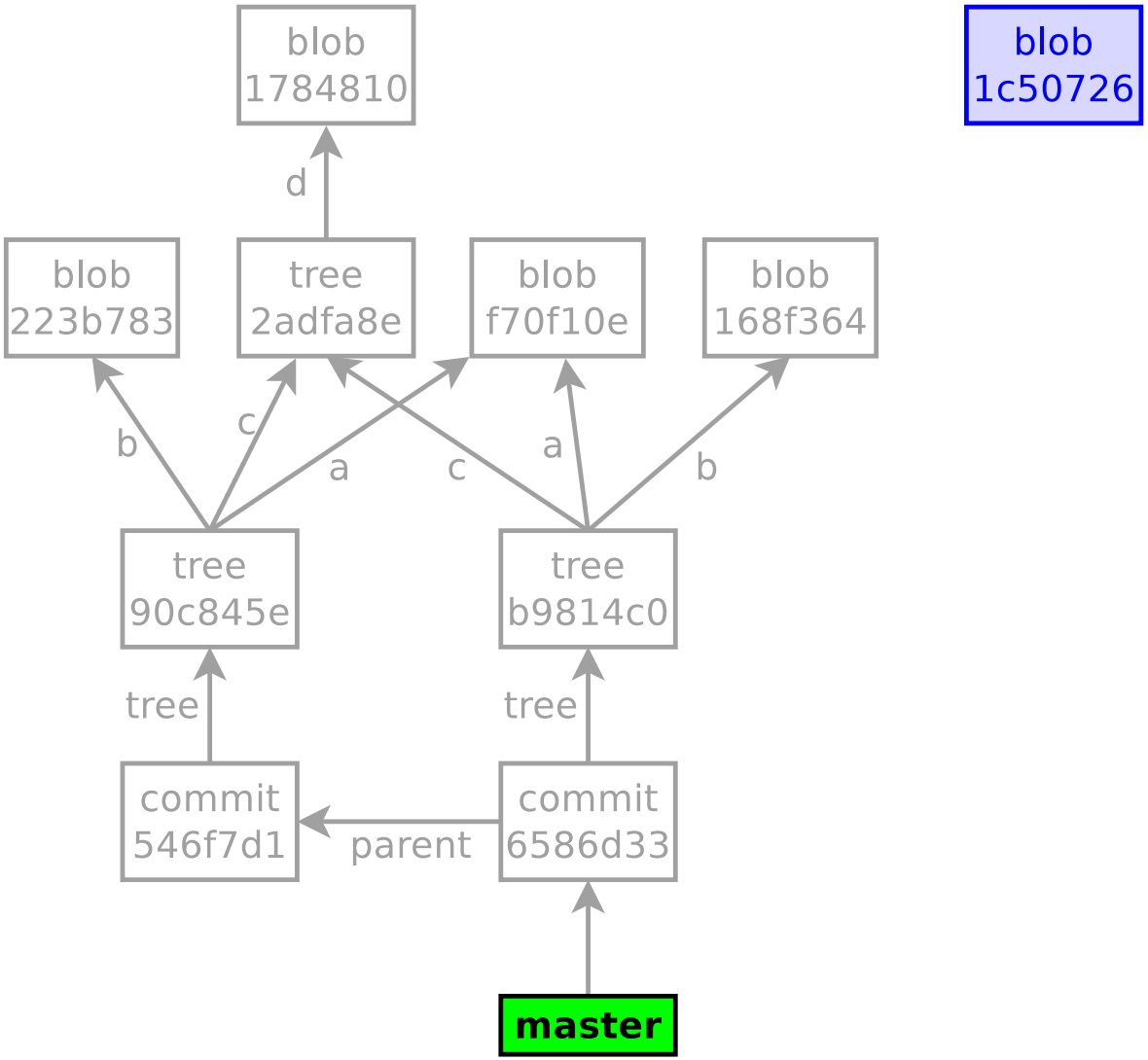
Git DB Example



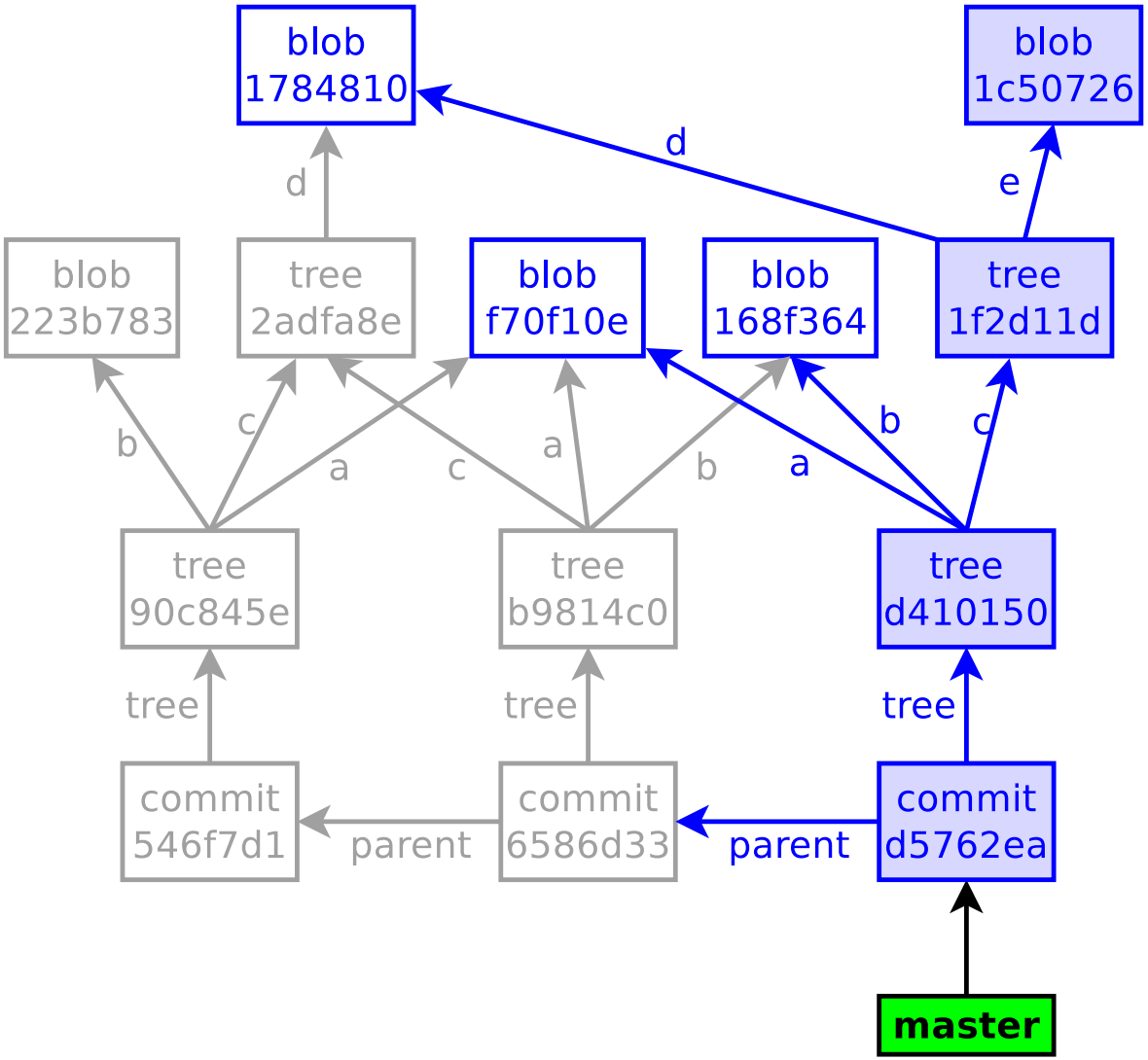
Git DB Example



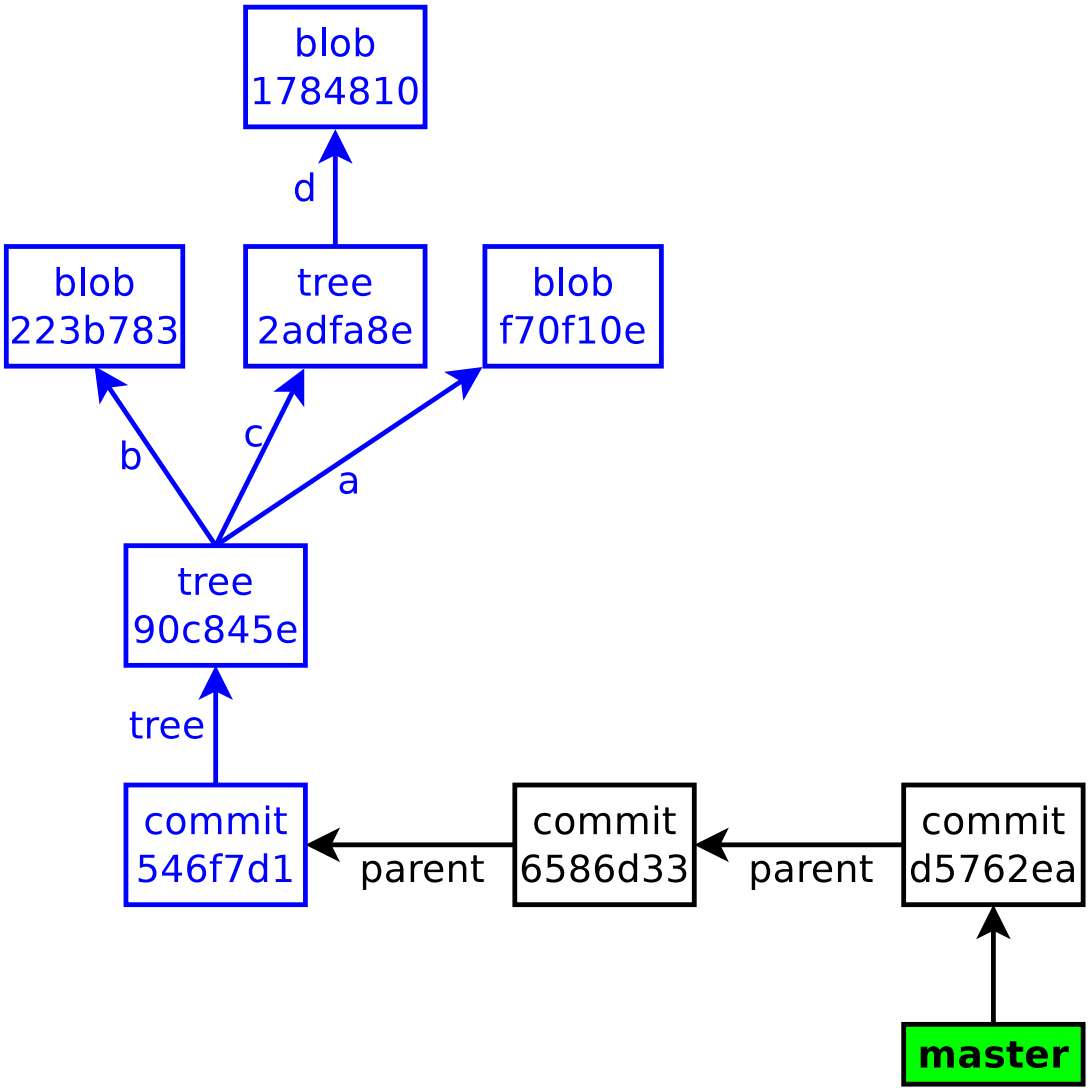
Git DB Example



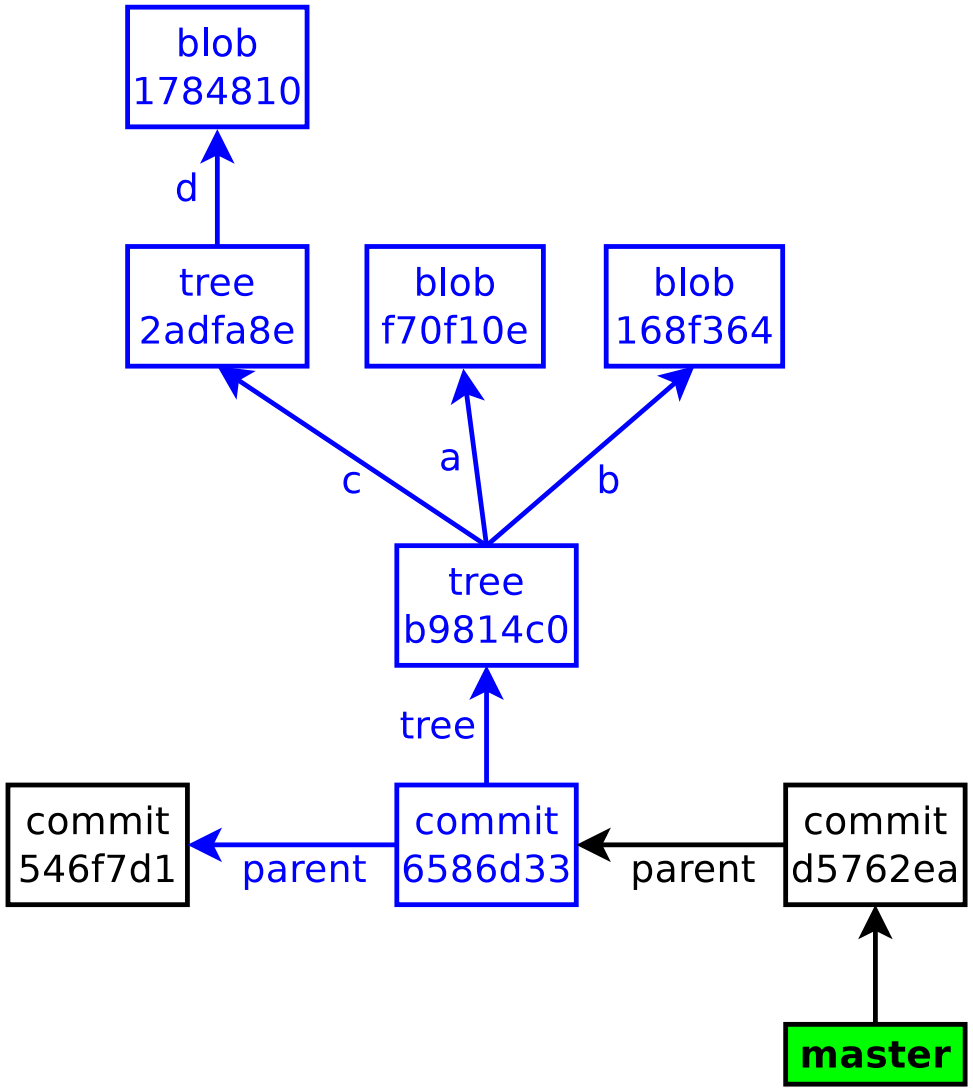
Git DB Example



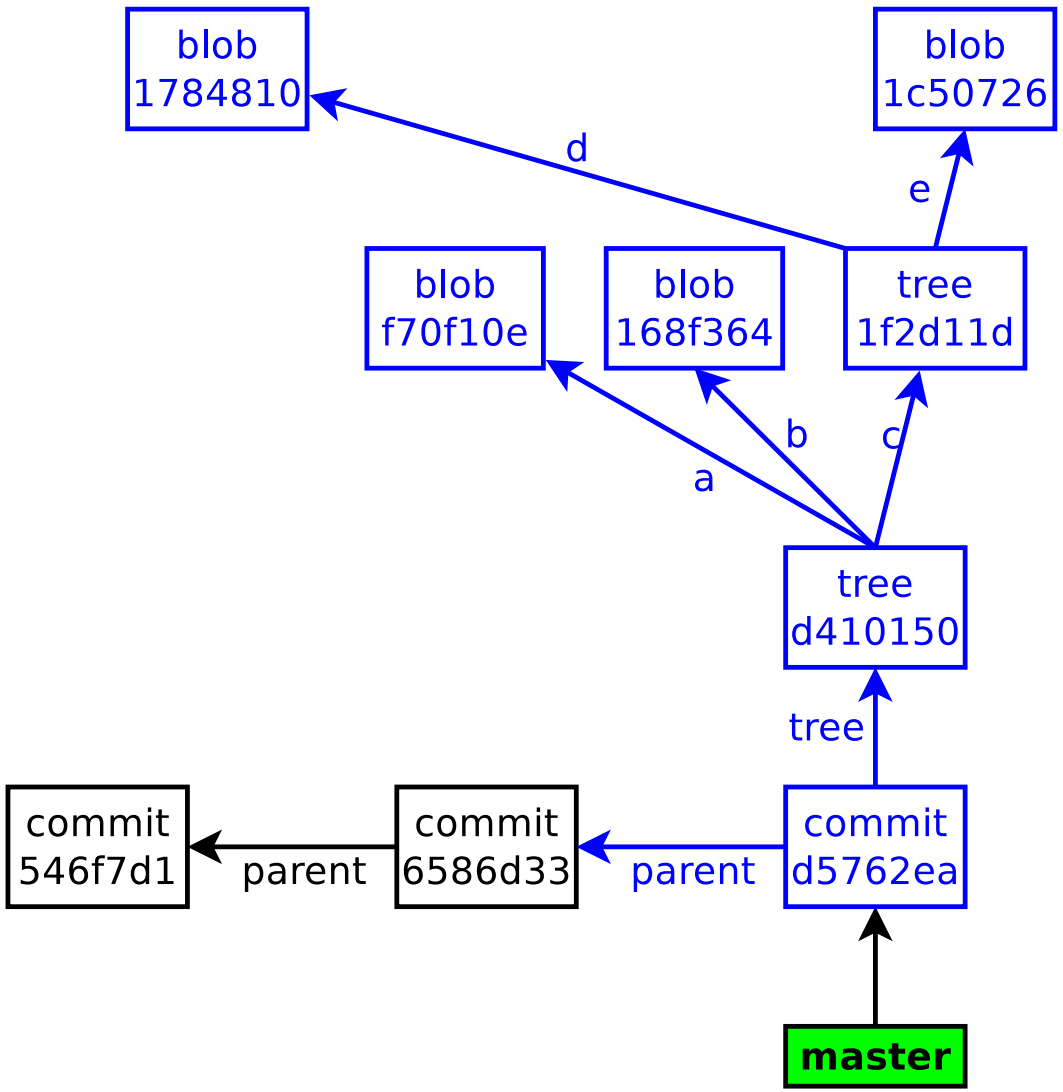
Git DB Example



Git DB Example



Git DB Example



Some low-level DB commands

`git cat-file` *type object*	read a DB entry
`git hash-object -w -t` *type file*	write a DB entry
`git ls-tree` *tree*	read a tree object (txt format)
`git mk-tree`	make a tree object (from txt format)
`git diff-tree` *tree1 tree2*	show the differences between two trees
`git commit-tree -p` *parent* `-m` *message tree*	write a new commit object
`git merge-tree` *base-tree left-tree right-tree*	prepare a 3-way merge

Git Refs

A **ref** is a pointer to a database object,[2] or possibly to another ref.

The 3 native types of refs are :

branch	→ `refs/heads/`*branch_name*
tag	→ `refs/tags/`*tag_name*
remote branch	→ `refs/remotes/`*remotename*`/`*branchname*

Refs are the entry point to the database. Git's garbage collector discards objects that are no longer reachable from a ref.

They may be stored in `.git/refs/` or in `.git/packed-refs`.

[2] usually a **commit** but it may also be a **tag** in case of annotated/signed tags

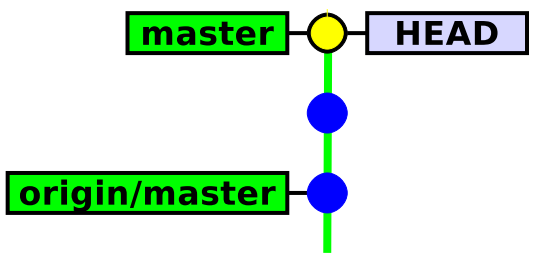
Special refs

`HEAD`	base of the working tree
`FETCH_HEAD`	upstream head from the last `git fetch`[3]
`ORIG_HEAD`	last `HEAD` before commands doing drastic changes (eg: `merge`, `rebase`)
`MERGE_HEAD`	commit being merged[3] *(when resolving a conflict)*
`CHERRY_PICK_HEAD`	commit being cherry-picked *(when resolving a conflict)*

Special refs are stored directly in `.git/`

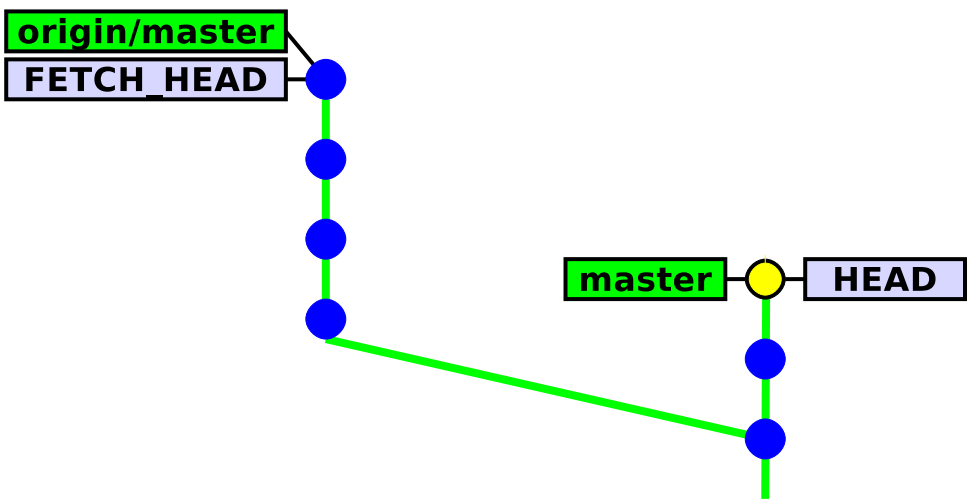
[3] `FETCH_HEAD` and `MERGE_HEAD` may refer to multiple commits

Special refs example



Special refs example

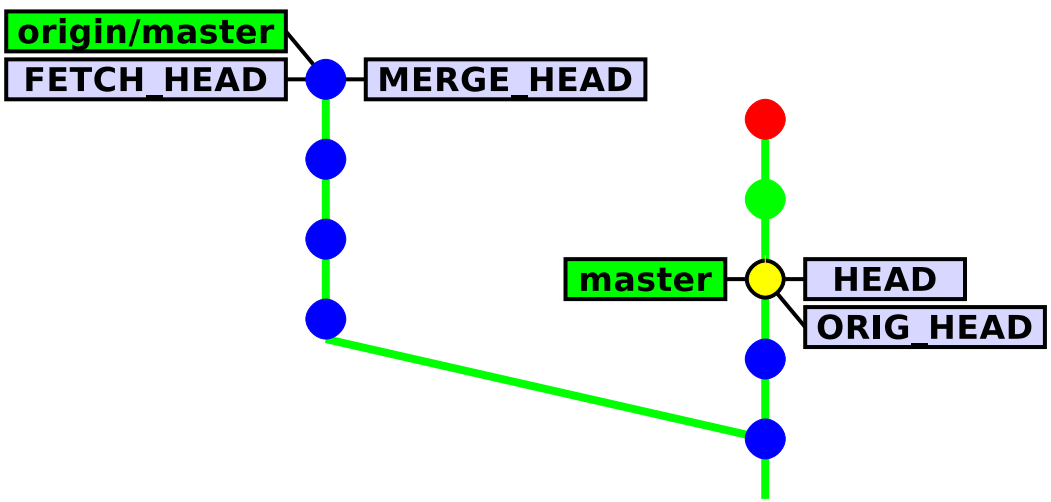
```
git fetch
```



Special refs example

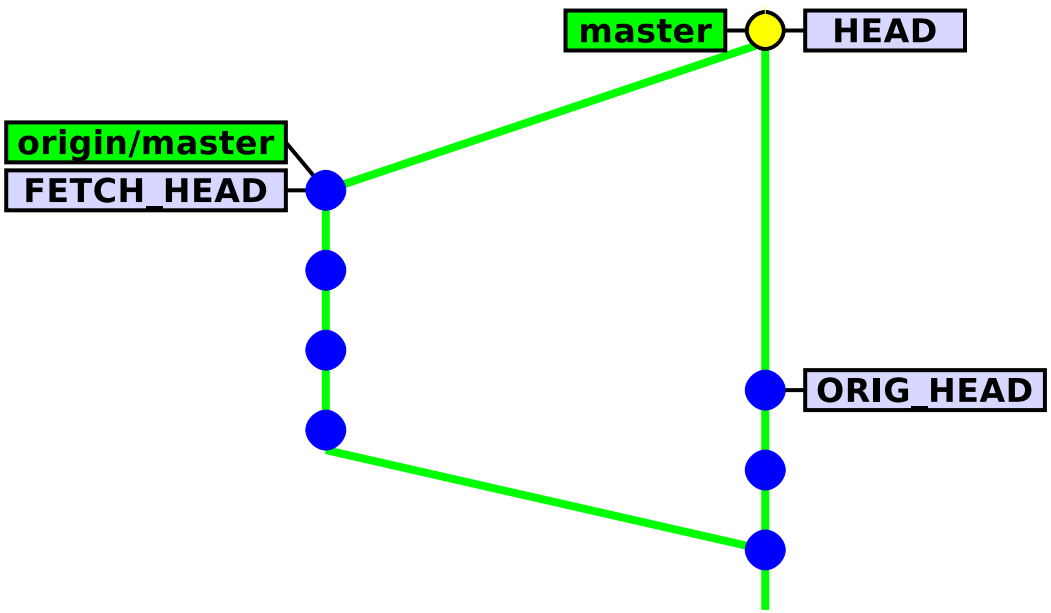
```
git merge origin/master
```

! ! conflict ! !



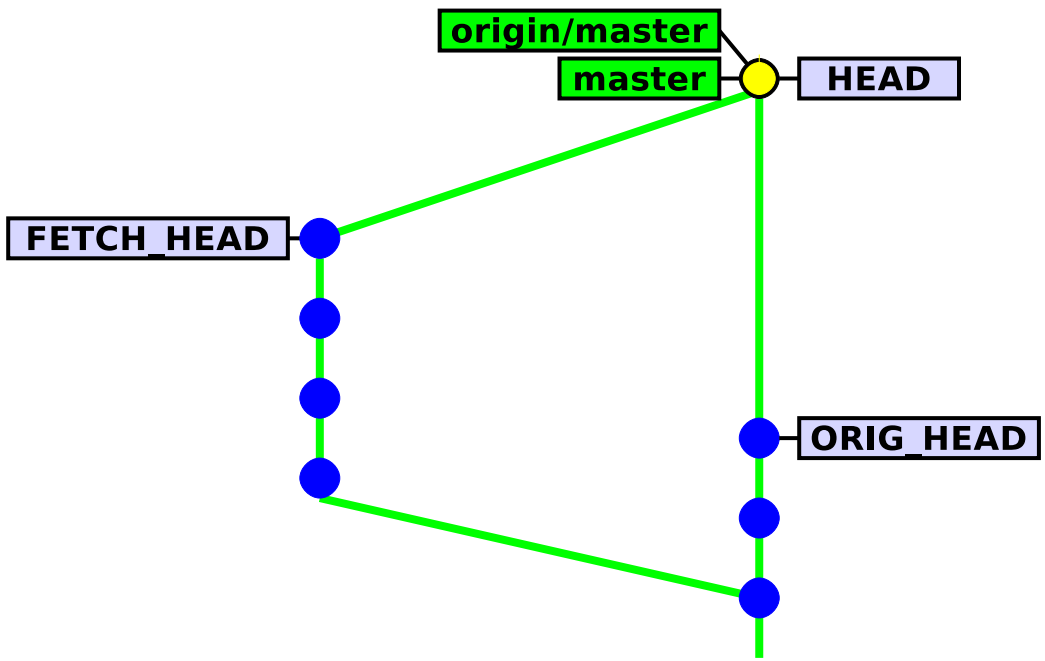
Special refs example

```
git commit
(finished merging)
```

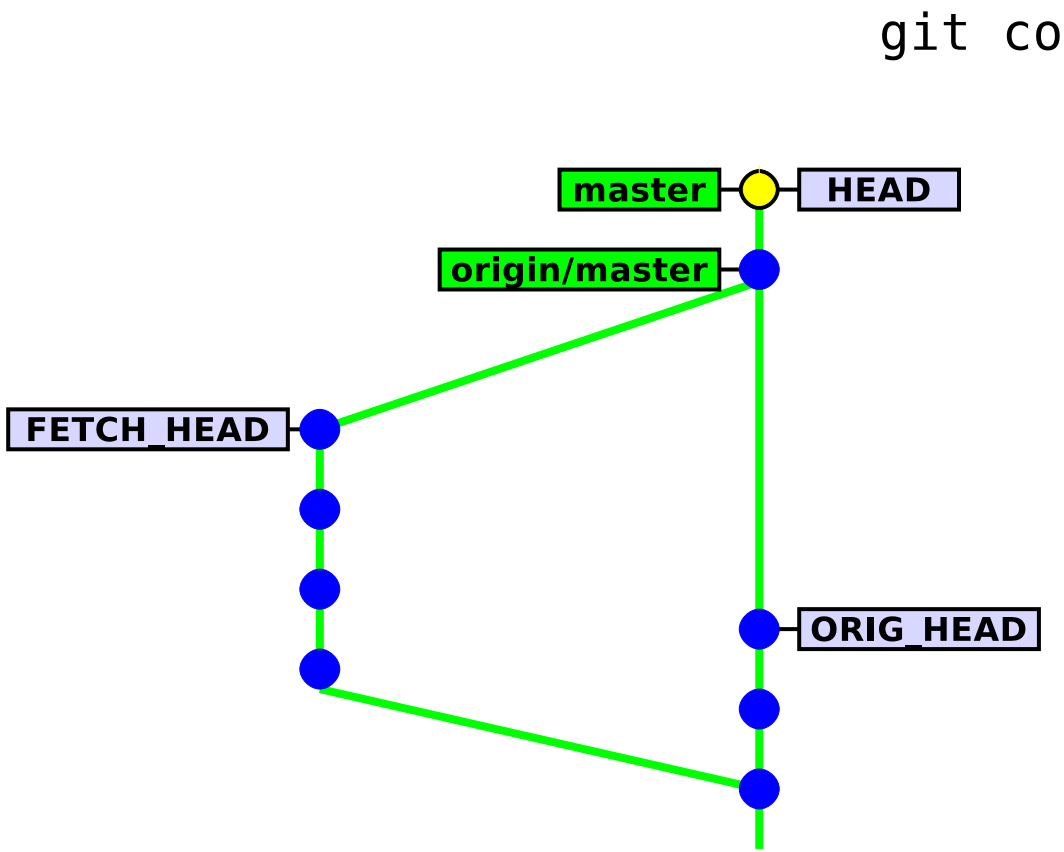


Special refs example

`git push`



Special refs example



Branch logs

`.git/logs/` stores the history of all branches (local and remote) and of the `HEAD` reference

A log line is created every time a command updates a branch to refer to a different commit (eg `commit checkout push` ...)

They can be browsed with `git reflog`

```
$ git reflog HEAD
f165ace HEAD@{0}: commit: hooks
13db5c9 HEAD@{1}: commit: repo layout
d74db73 HEAD@{2}: commit: fixed headers
16180f1 HEAD@{3}: commit: use FETCH_HEAD directly
  :       :          :
adb1fb6 HEAD@{16}: merge 810053ec98502f04943c06f6ca3bfaf3d3f2060d: Merge made by the 'recursive'
strategy.
  :       :          :
77d7f39 HEAD@{81}: checkout: moving from master to 77d7f39
```

Hooks

- Hooks are customisable commands that are automatically executed by git when performing specific actions
- They are typically used to trigger external behaviours *(eg. run a continuous integration build in the* `post-receive` *hook)*
- They are stored in `.git/hooks/`

Hooks list

`applypatch-msg`
`prepare-commit-msg`
`commit-msg`
	customise the default commit messages

`pre-applypatch` `post-applypatch`	before/after applying a patch
`pre-commit` `post-commit`	before/after `commit`

`post-checkout`
`pre-rebase`
`post-merge`
`post-rewrite`
	after `checkout`
	before `rebase`
	after `merge`
	after rewriting the history

`pre-receive`
`update`
`post-receive`
`post-update`
	before/during/after `push` (server-side)

`pre-auto-gc`	before automatic garbage collection (`gc`)

Part 3.
Specifying revisions

Specifying revisions (1/4)

`man gitrevisions`

There are many ways to reference a particular commit in a git command

- sha1sum of the commit (or the first digits if not ambiguous)
  `dae86e1950b1277e545cee180551750029cfe735`
  `dae86e1`
  `dae8`

- output of the `git describe` command
  `v1.7.4.2-679-g3bee7fb`

Specifying revisions (2/4)

`man gitrevisions`

- a reference (listed by lookup order)
  1. special ref (`.git/*`)
     `HEAD`
  2. raw reference (`.git/refs/*`)
     `heads/master`
     `tags/v1.0`
     `remotes/origin/master`
     `remotes/origin/HEAD`
  3. tag (`.git/refs/tags/*`)
     `v1.0`
  4. local branch (`.git/refs/heads/*`)
     `master`
  5. remote branch (`.git/refs/remotes/*`)
     `origin/master`
  6. HEAD of a remote repository (`.git/refs/remotes/*/HEAD`)
     `origin`

Specifying revisions (3/4)

`man gitrevisions`

- lookup in your **local** logs (`.git/logs/*`)
  - by date:
    `@{yesterday}`
    `@{5 minutes ago}`
    `master@{2013-09-10}`
  - by n-th prior position:
    `@{1}` → immediate prior position of the current branch
    `@{5}` → 5th prior position
    `master@{1}`
  - by n-th previous `git checkout`
    `@{-1}` → last checked out branch
    `@{-5}` → 5th last checked out branch

Specifying revisions (4/4)

`man gitrevisions`

- upstream branch
  `master@{upstream}` → usually `origin/master`

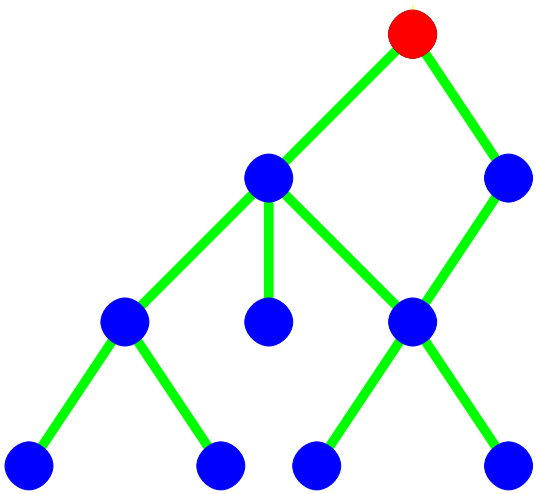
- n-th parent of a commit
  `HEAD^1` or `HEAD^` → first parent
  `HEAD^2` → second parent (in case of `merge`)

- n-th generation ancestor (by following the first parent)
  `HEAD~1` → first parent (same as `HEAD^1`)
  `HEAD~2` → grand parent (same as `HEAD^1^1`)

- textual search in the parent commits messages
  `HEAD^{/bugfix}`

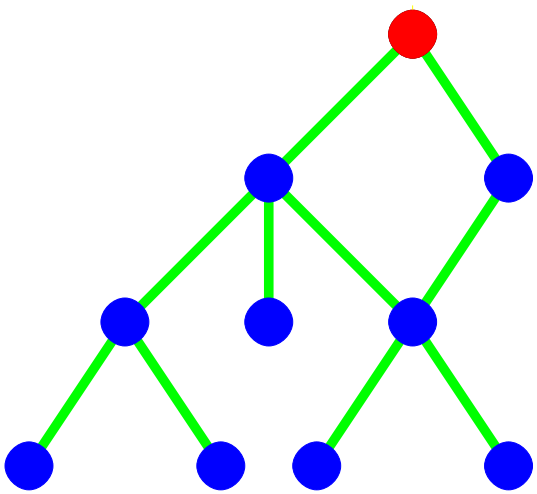
- regex search
  `:/bug ?fix` → message matching /bug ?fix/
  `:/!bug ?fix` → message not matching /bug ?fix/

Combinations



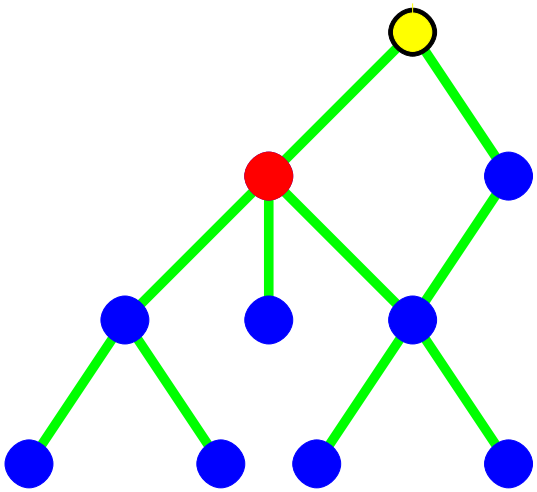
HEAD

Combinations



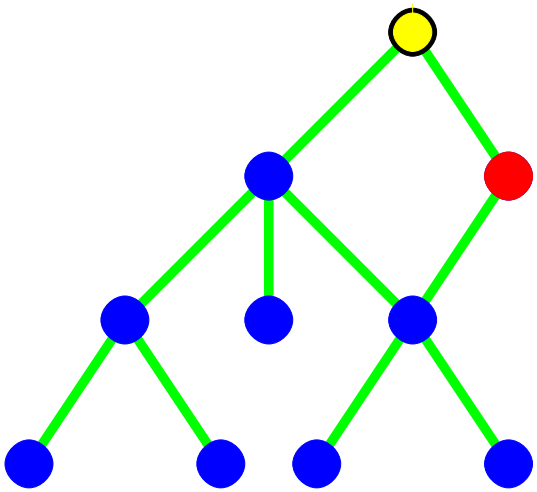
`HEAD^0`

Combinations



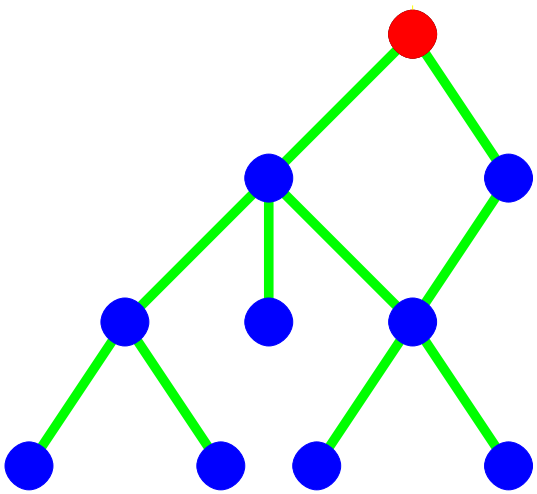
```
HEAD^1
HEAD^
```

Combinations



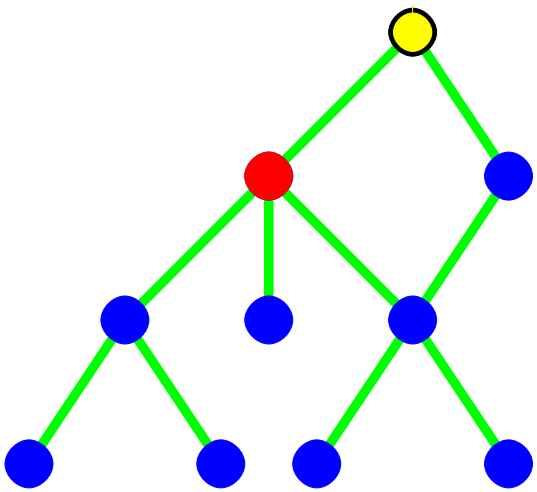
HEAD^2

Combinations



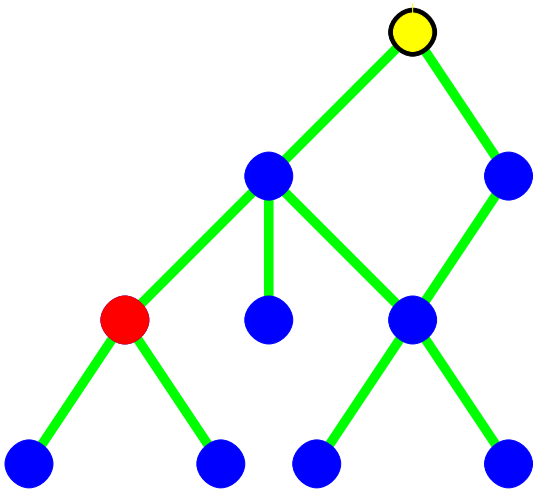
```
HEAD~0
```

Combinations



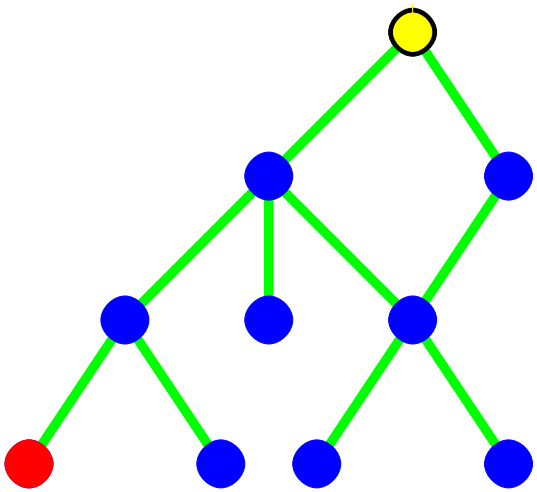
```
HEAD~1
HEAD~
```

Combinations



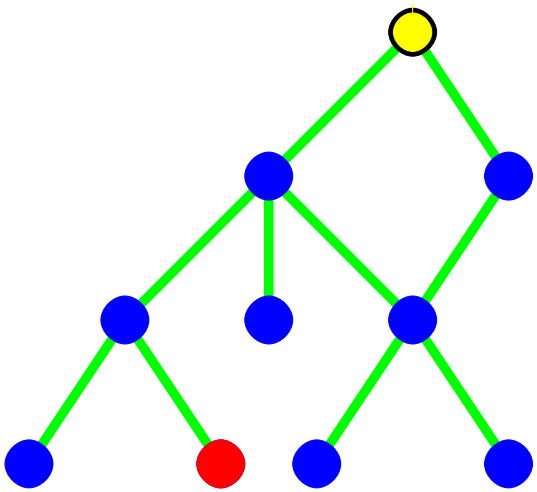
```
HEAD~2
HEAD~~
HEAD^1^1
HEAD^^
```

Combinations



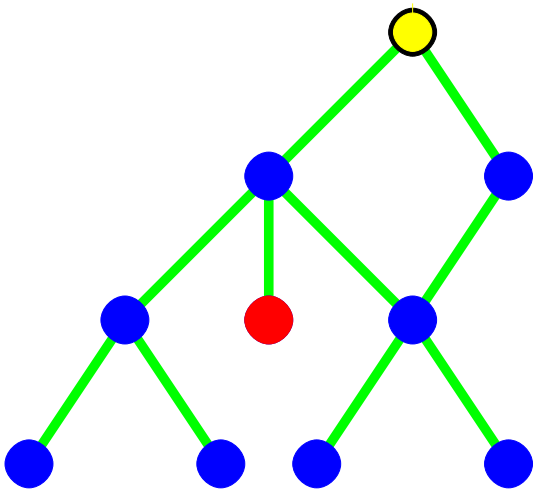
```
HEAD~3
HEAD~~~
HEAD^1^1^1
HEAD^^^
HEAD^1~^
```

Combinations



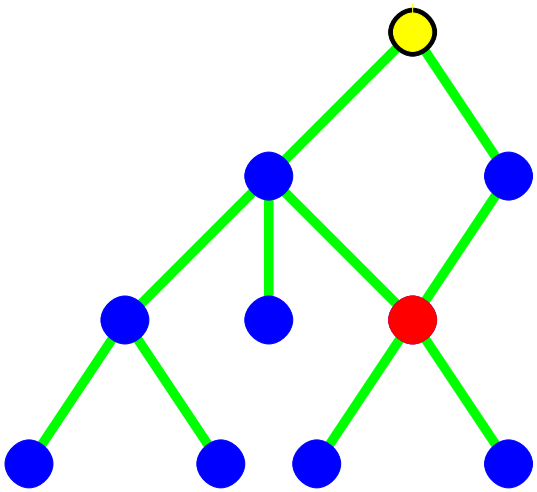
```
HEAD~2^2
HEAD^^^2
```

Combinations



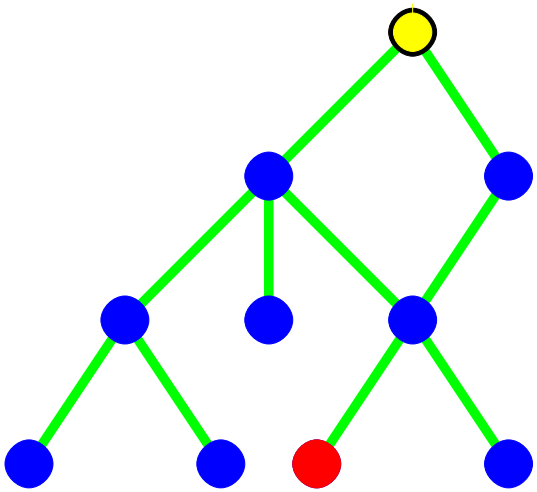
```
HEAD~^2
HEAD^^2
```

Combinations



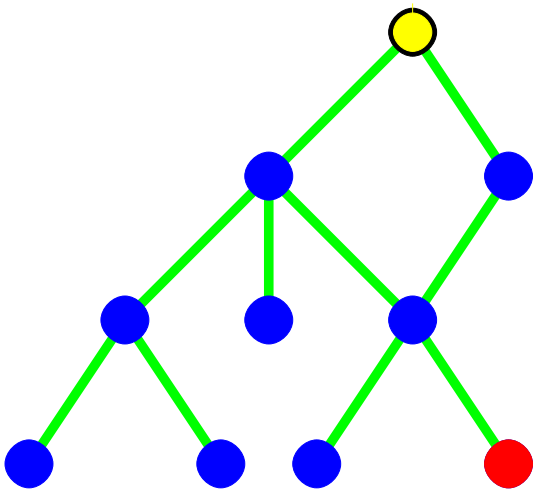
```
HEAD^^3
HEAD~1^3
HEAD^2~
HEAD^2^1
```

Combinations



```
HEAD^^3^
HEAD^2^^
HEAD^2~2
```

Combinations



```
HEAD^^3^2
HEAD^2^^2
```

Specifying other objects

Some commands (`git show`, `git cat-file` ...) accept other kinds of objects (tag, tree, blob)

- by object type
  `master^{tree}` → root tree
- by path
  `HEAD:README`
  `HEAD~2:some/file/in/the/tree`
- by path (relative to the current directory)
  `HEAD:./somefile`

Objects from the index

- staged file
  `:0:README`
- other stages (during a `git merge`)
  `:1:README` → common ancestor
  `:2:README` → target branch (typically the current branch)
  `:3:README` → version being merged

Part 4.
Playing with your index

Saving/restoring the index & working copy

```
git stash [-u]
git stash [-u] save message
```

This command:

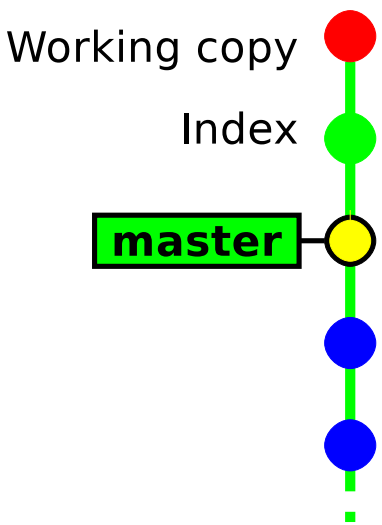
1. saves the current state of the index and the working copy
2. resets them (`git reset --hard`)

With `-u`, untracked files are stashed too.

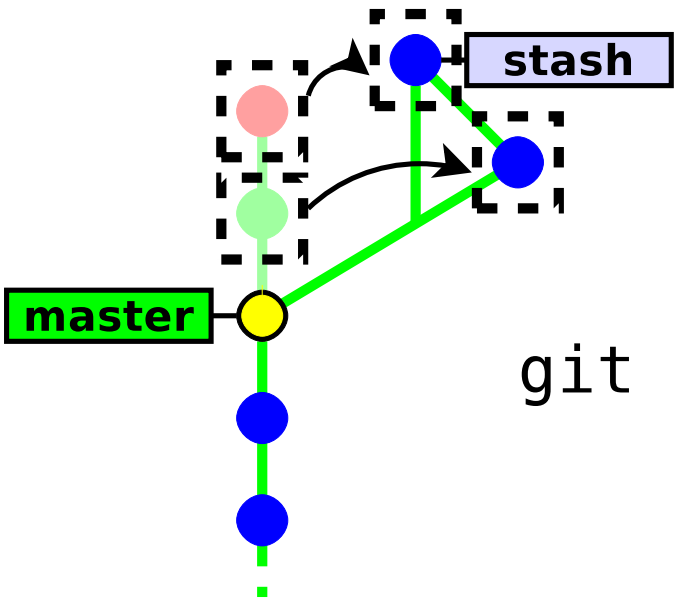
```
git stash pop
```

→ apply the stashed changes into the current working tree

Stash example

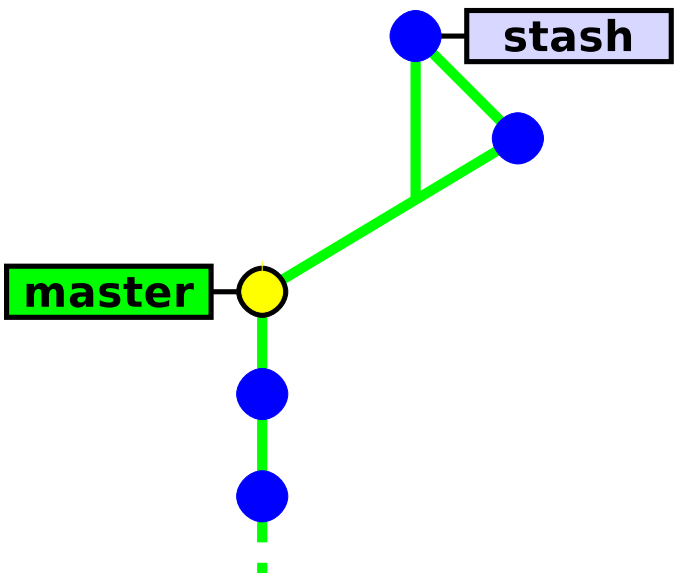


Stash example

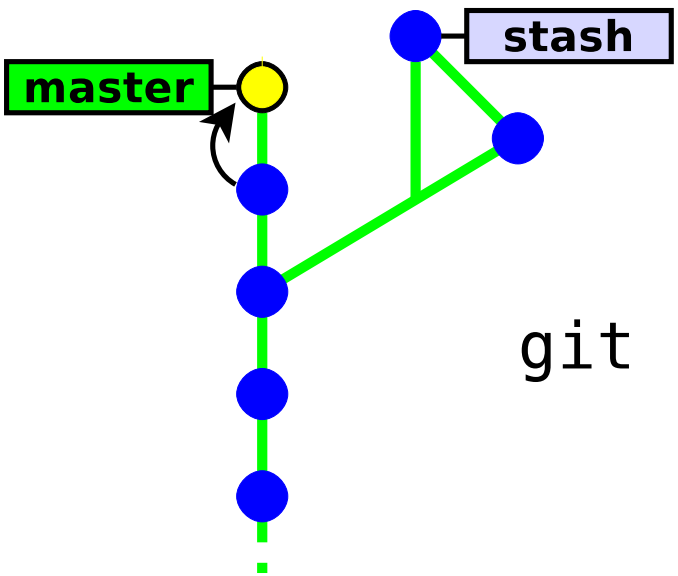



```
git stash
```

Stash example

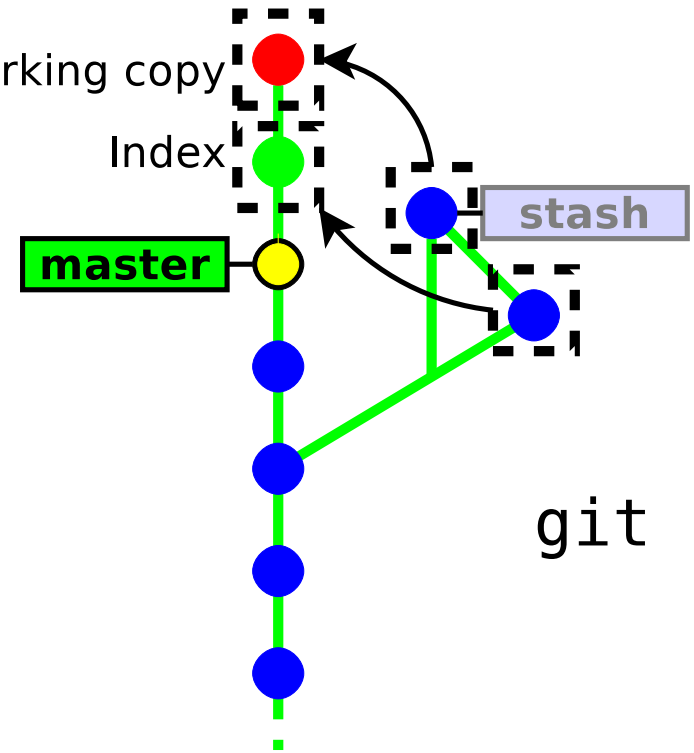


Stash example



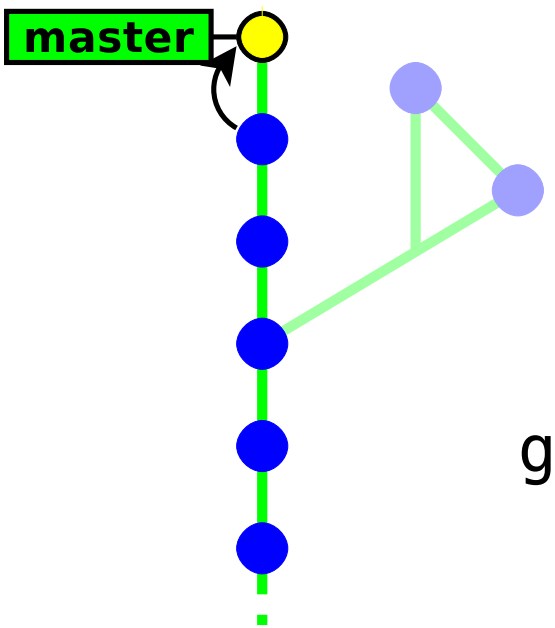

```
git commit
```

Stash example




```
git stash pop
```

Stash example




```
git commit
```

Multiple stashes

You can stash multiple times. Changes are stored in a LIFO stack

→ `git stash pop` works with the latest changeset by default

```
git stash list
```

→ list all stashed changes

```
git stash pop stash@{id}
```

→ apply an arbitrary changeset

`stash@{0}` is the latest changeset, `stash@{1}` the previous one...

Staging patches

```
git add -p [ file1 ... ]
```

Instead of adding the whole file, `git add -p` walks through the changes and let you select which hunk you want to add.

```
git add -e [ file1 ... ]
```

`git add -e` launches an editor and let you edit the patch that will be applied to the index

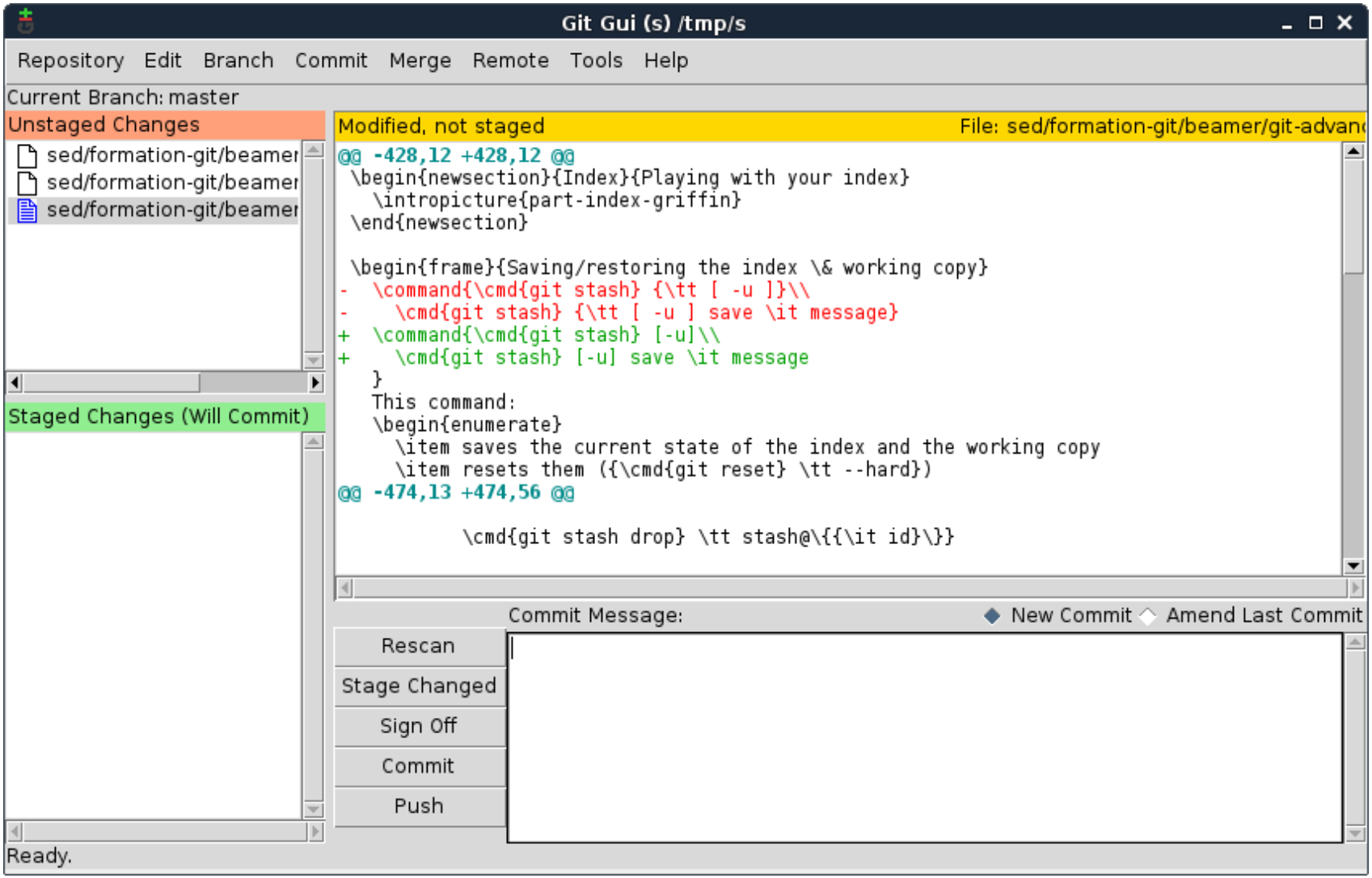
Example: git add -p

```
$ git add -p
index 7b61025..2617fcf 100644
--- a/sed/formation-git/beamer/git-advanced.tex
+++ b/sed/formation-git/beamer/git-advanced.tex
@@ -476,6 +476,13 @@
   {\tt stash@\{0\}} is the latest change, {\tt stash@\{1\}} the previous one\ldots
\end{fancyframe}

+\begin{frame}{Staging patches}
+  \command{\cmd{git add -p}  [{\it~file1  \ldots~}]}
+
+  Instead of adding the whole file, \cmd{git add -p} walks through the changes
+  and let you select which hunk you want to add.
+\end{frame}
+
 % non-linear development
 % - stash
 % - partial commits
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

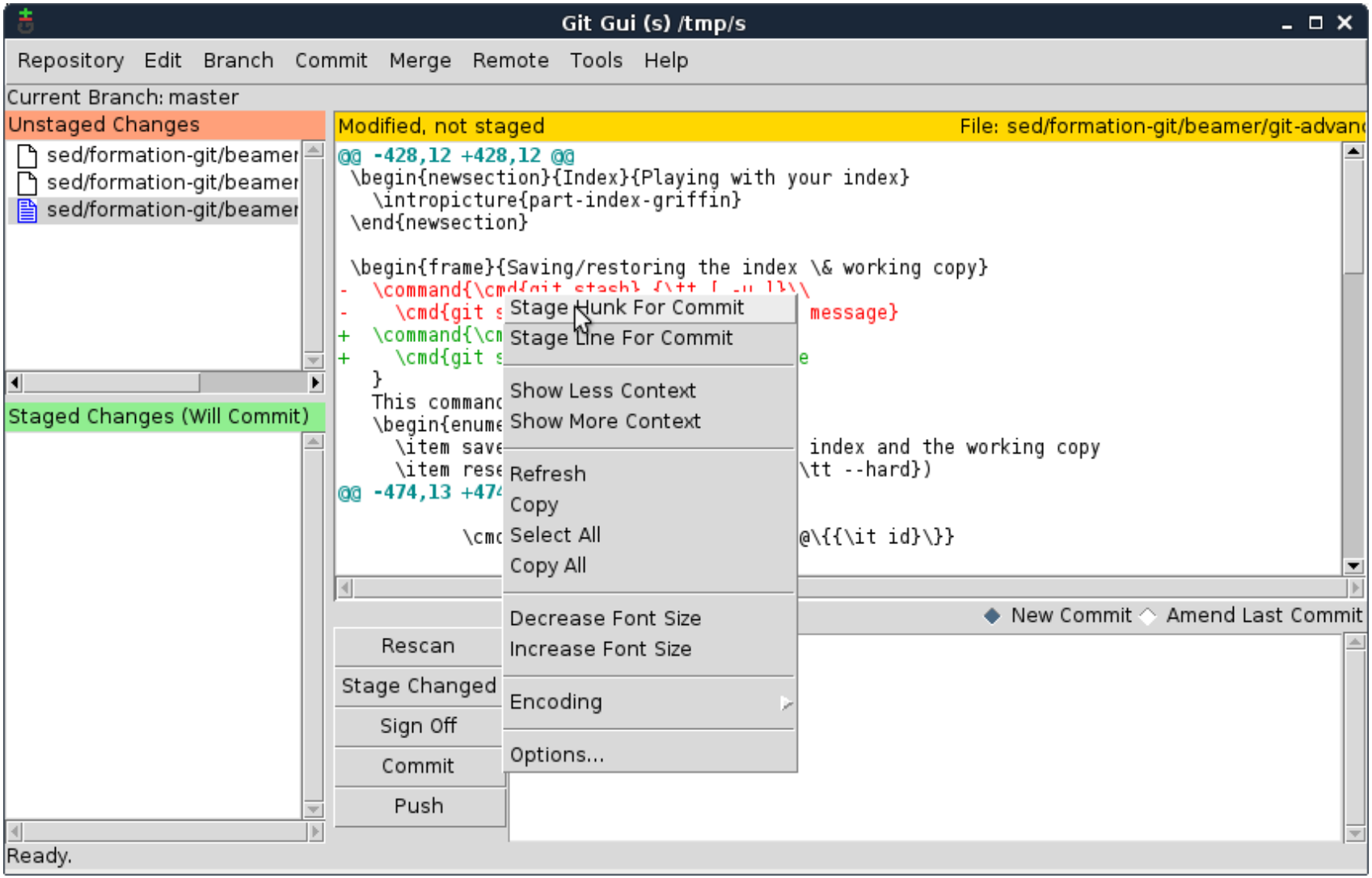
Example: `git add -e`

```
diff --git a/sed/formation-git/beamer/git-advanced.tex b/sed/formation-git/beamer/git-advanced.tex
index 7b61025..ca0ea98 100644
--- a/sed/formation-git/beamer/git-advanced.tex
+++ b/sed/formation-git/beamer/git-advanced.tex
@@ -472,16 +472,55 @@

           \cmd{git stash drop} \tt stash@\{{\it id}\}}

   \ra apply/drop an arbitray change\\
   {\tt stash@\{0\}} is the latest change, {\tt stash@\{1\}} the previous one\ldots
 \end{frame}

+
+\begin{frame}[fragile]{Staging patches}
+  \command{\cmd{git add -p}  [{\it~file1  \ldots~}]}
+
+  Instead of adding the whole file, \cmd{git add -p} walks through the changes
+  and let you select which hunk you want to add.
+
+  ~
+
+  \command{\cmd{git add -e}  [{\it~file1  \ldots~}]}
+
+  \cmd{git add -e} launches an editor and let you edit the patch that will be
+  applied to the index
+
+\end{frame}
+
+\begin{frame}[fragile]{Example: \cmd{git add -p}}
                                                              1,1           Haut
```

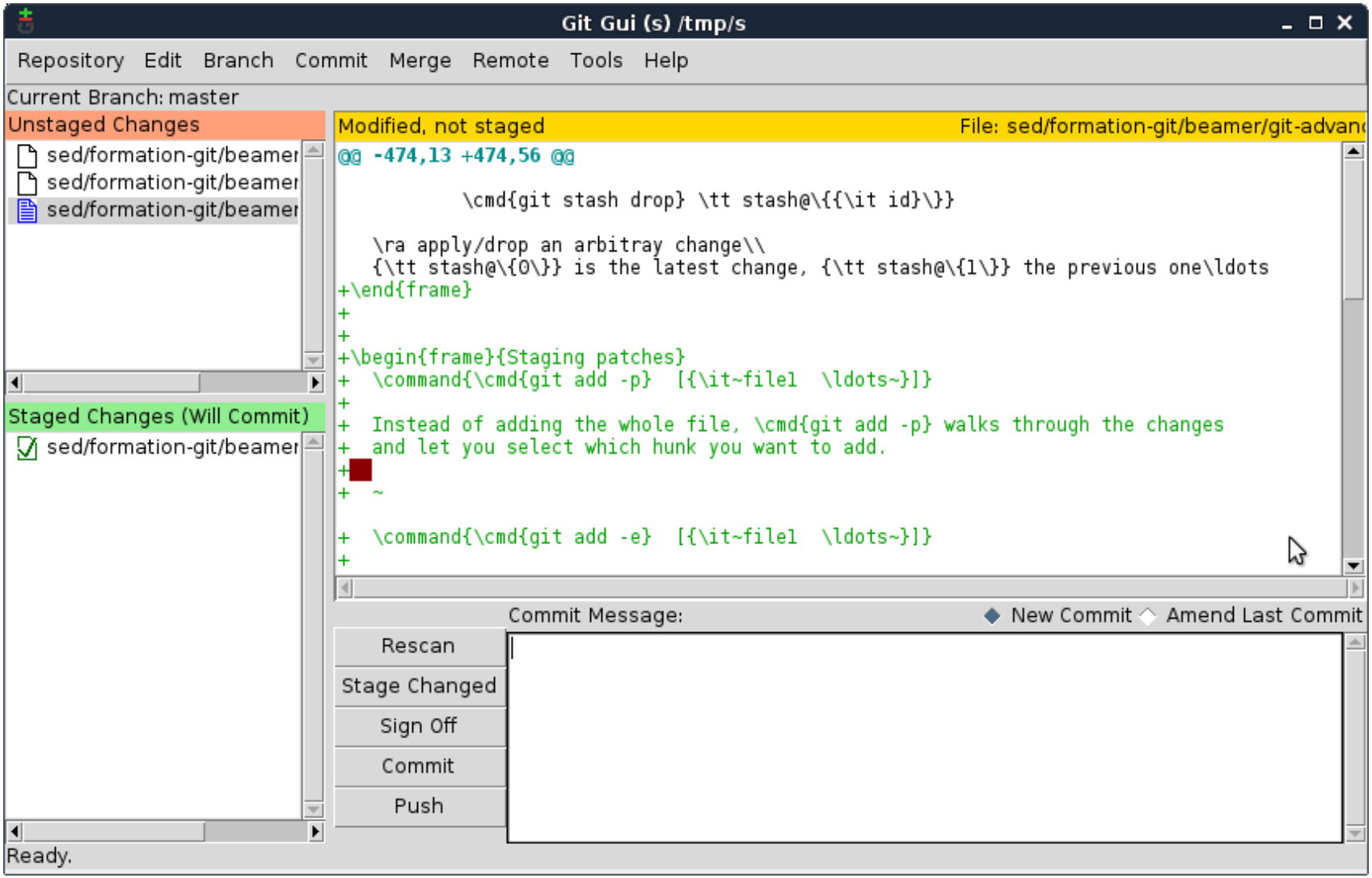
Staging hunks/lines in `git gui`



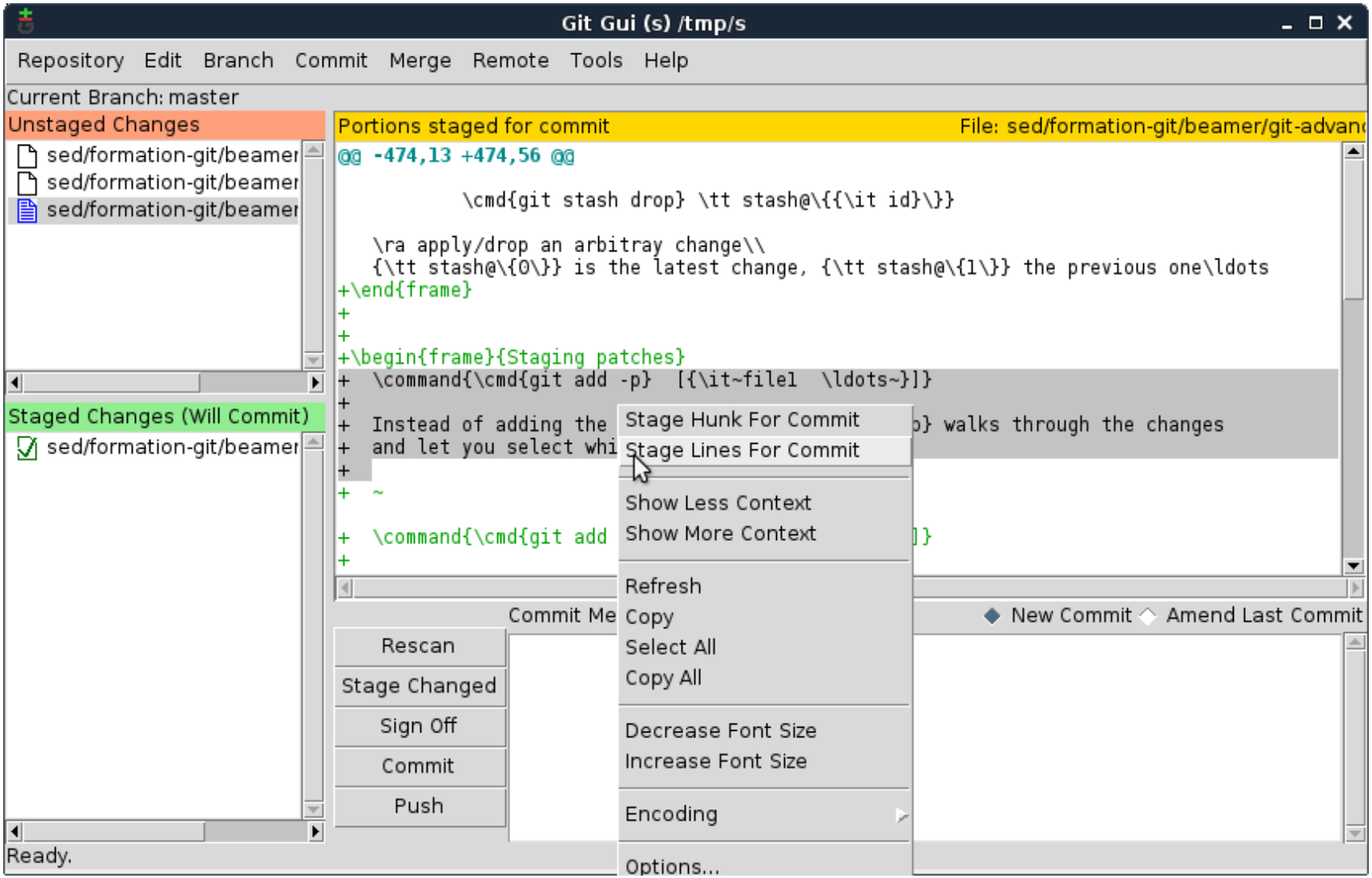
Staging hunks/lines in `git gui`



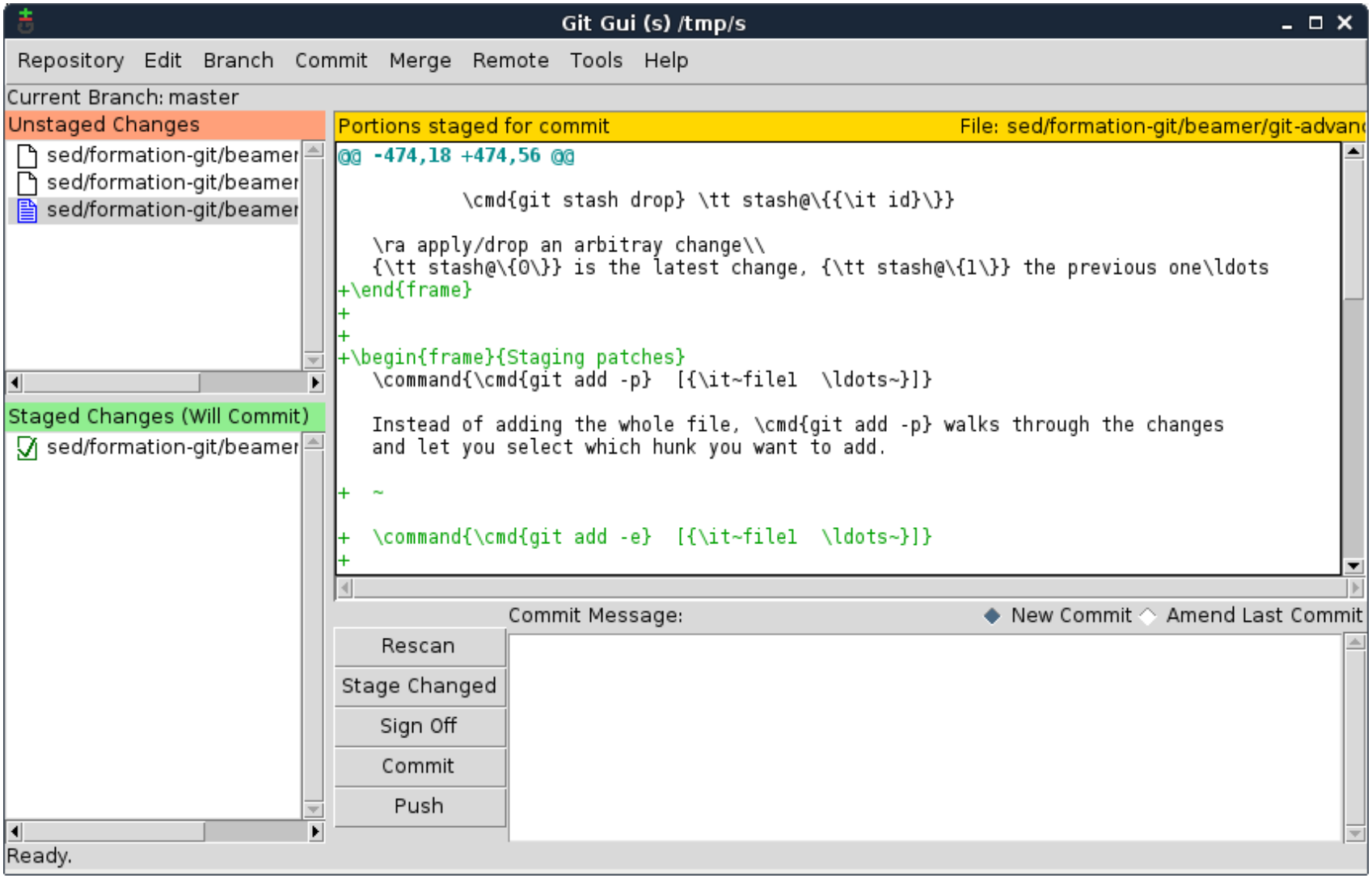
Staging hunks/lines in `git gui`



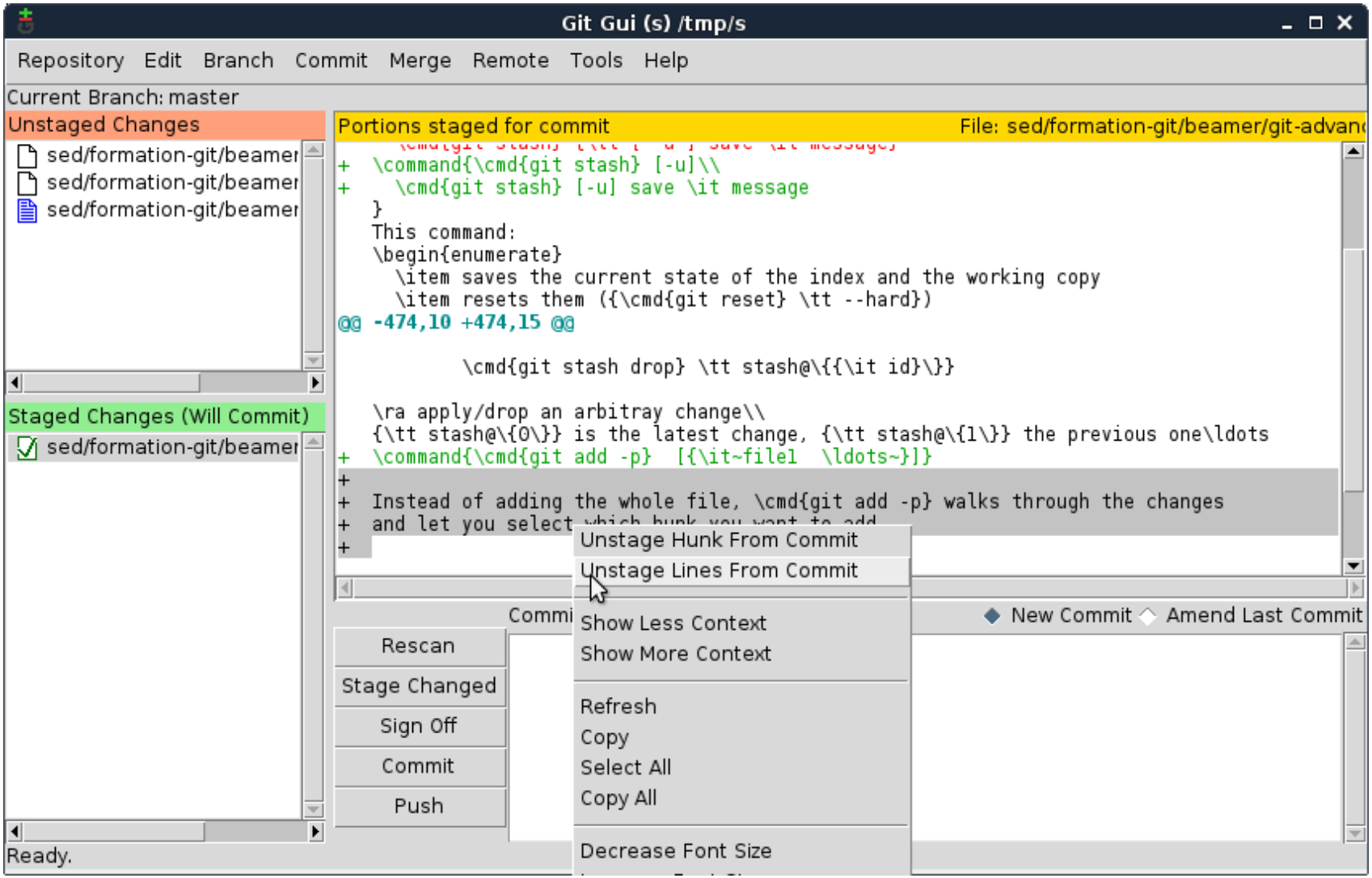
Staging hunks/lines in `git gui`



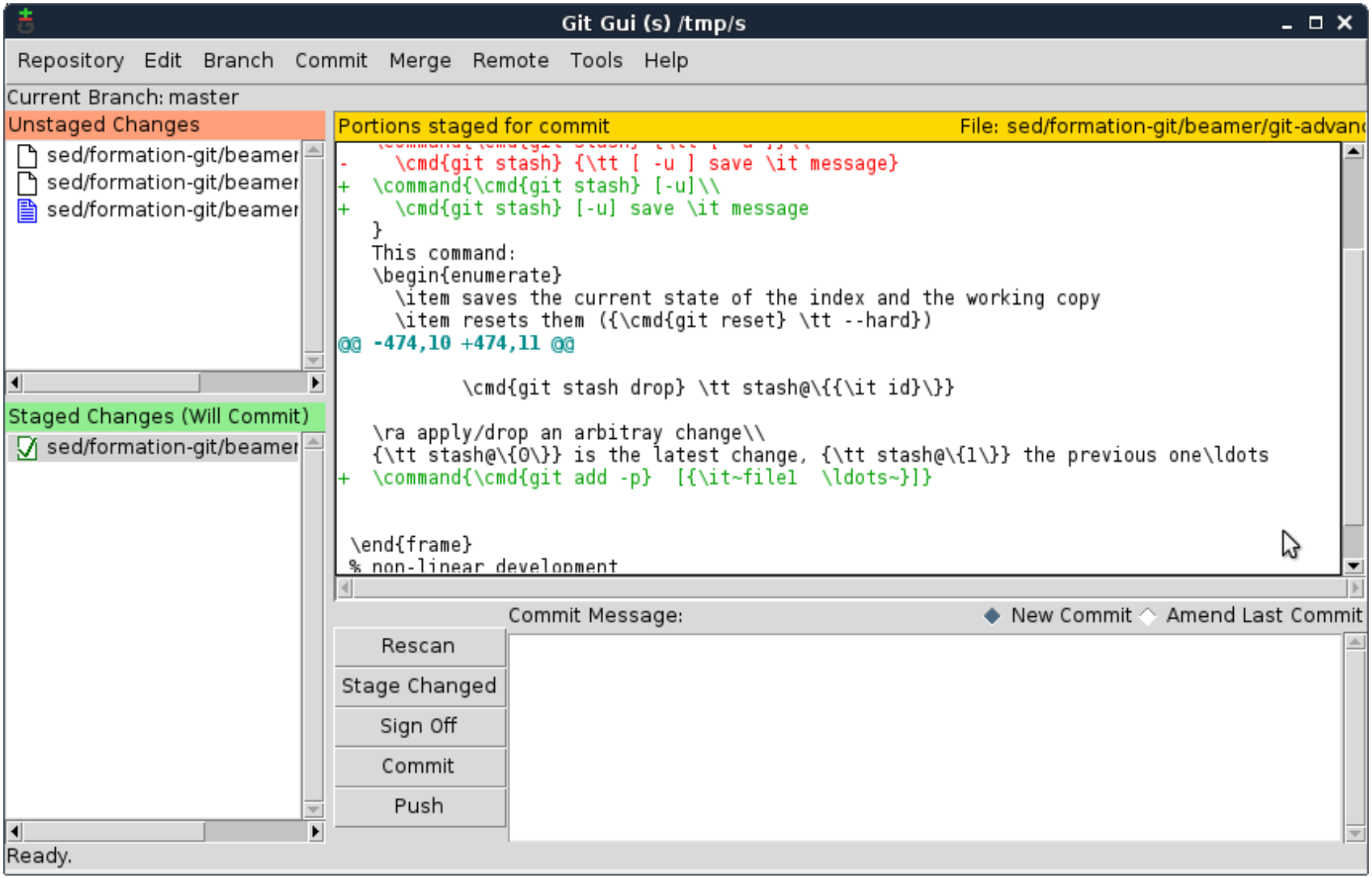
Staging hunks/lines in `git gui`



Staging hunks/lines in `git gui`



Staging hunks/lines in `git gui`



Part 5.
Rewriting your history

Reasons to rewrite your history

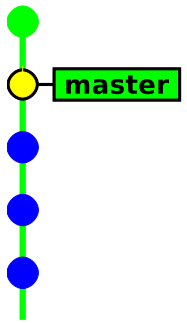
- fix an omission in the last commit
- linearise the history
- clean a branch before pushing
- rollback
- discard content that must not be there
- ~~hide your mistakes~~

Amending the the last commit

```
git commit --amend [...]
```

Committing with `--amend` makes git meld the new commit with the previous commit.

```
$ git commit -m "some changes"
.. oops! one file is missing ..
$ git add new_file
$ git commit --amend
```



Note:

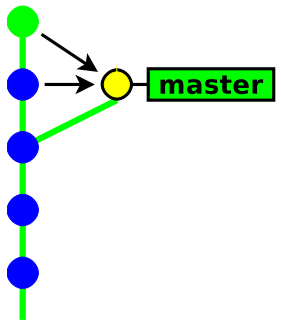
- the original commit is discarded
- the new commit has a different sha1sum (different content)

Amending the the last commit

```
git commit --amend [...]
```

Committing with `--amend` makes git meld the new commit with the previous commit.

```
$ git commit -m "some changes"
.. oops! one file is missing ..
$ git add new_file
$ git commit --amend
```



Note:

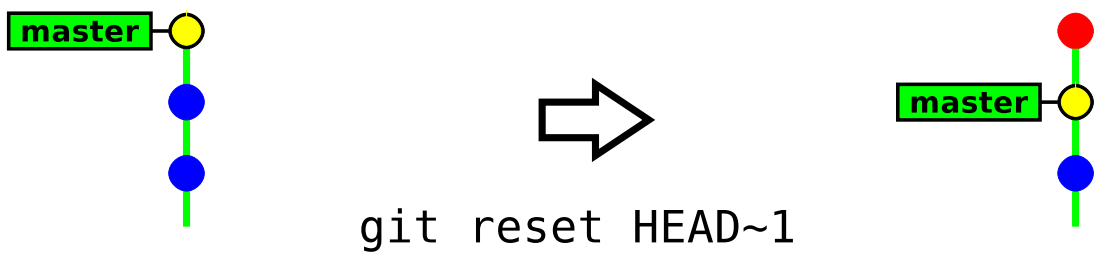
- the original commit is discarded
- the new commit has a different sha1sum (different content)

Rollback

```
git reset [ --hard ] refspec
```

`git reset` called with a *refspec* unconditionally moves the current branch head to *refspec*.

With `--hard`, the working copy is reset as well.

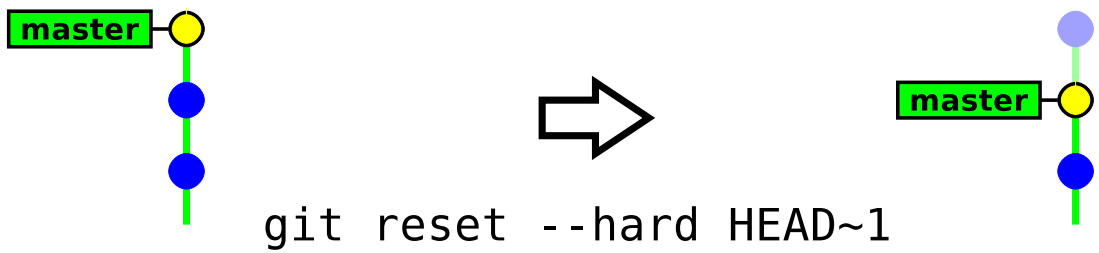


Rollback

```
git reset [ --hard ] refspec
```

`git reset` called with a *refspec* unconditionally moves the current branch head to *refspec*.

With `--hard`, the working copy is reset as well.



Rebasing a branch

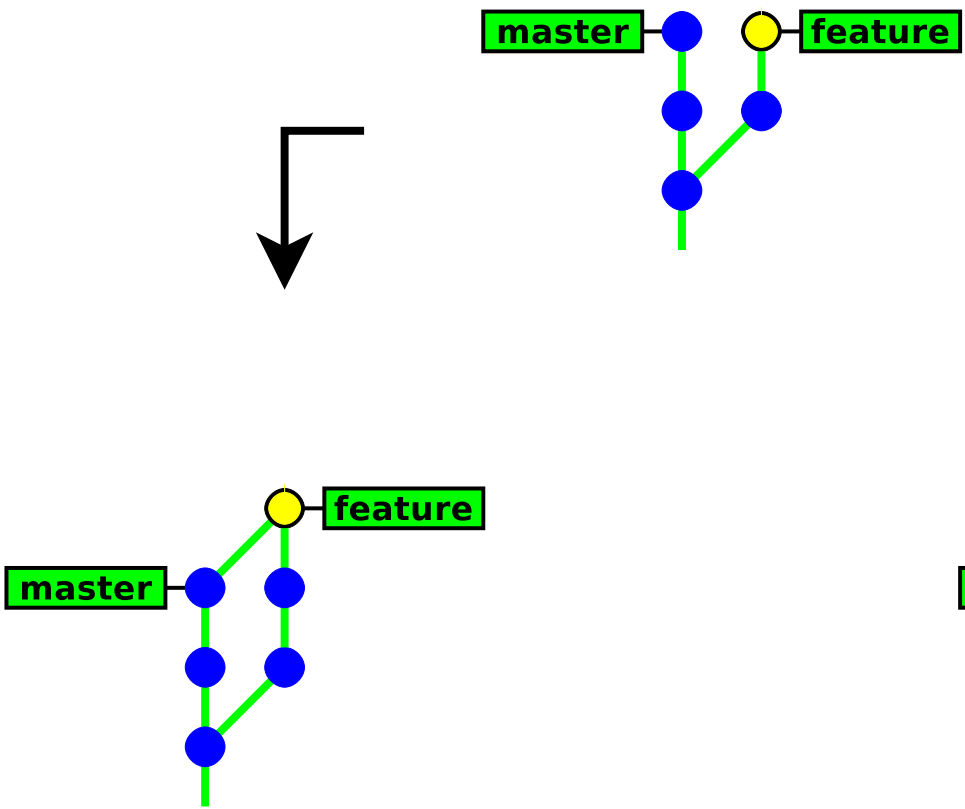
```
git rebase [ --onto newbase ] upstream
```

This command,

1. selects all commits since the last common ancestor with the *upstream* commit
2. *cherry-picks* these commits onto *upstream*[4]
3. moves the current branch to this new head
   (the previous branch is discarded)

[4] or onto *newbase* if given

git merge vs. git rebase



git merge master

git rebase master

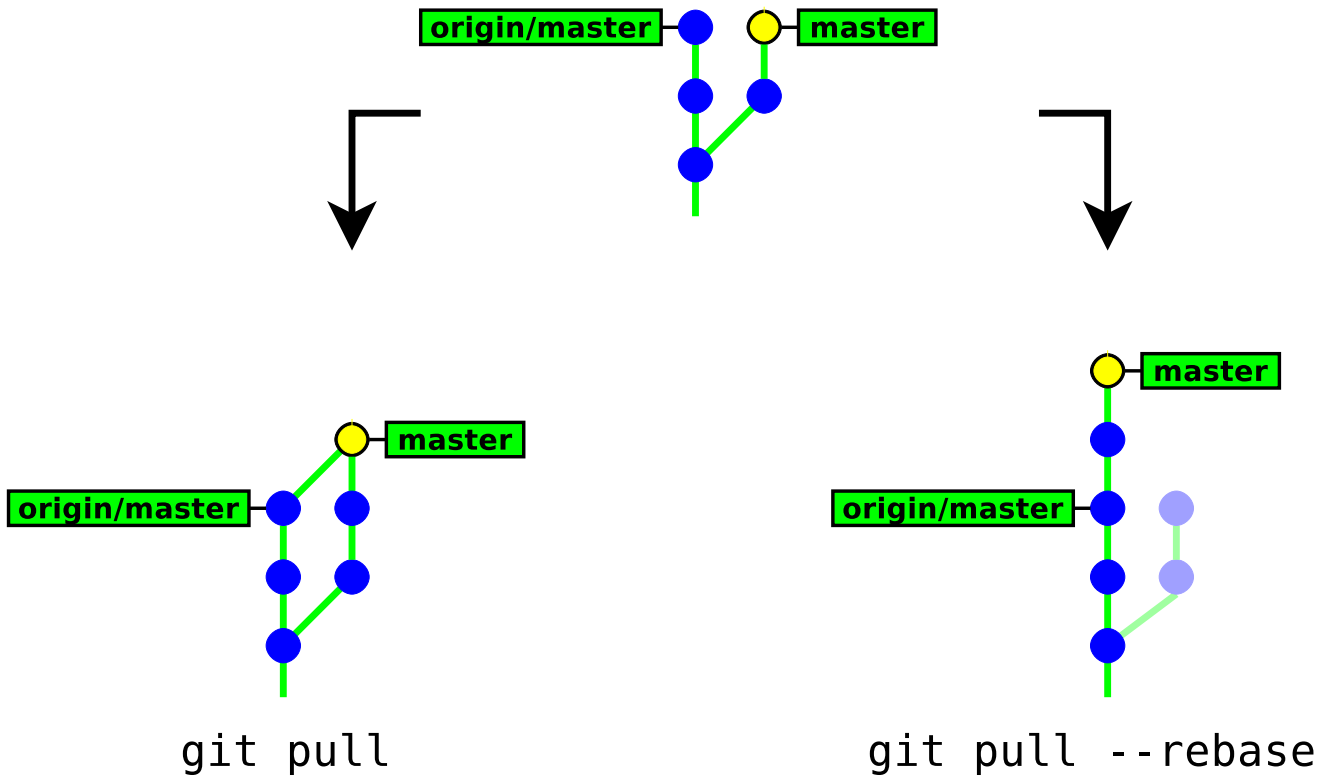
Limitations of `git rebase`

- The merge history and conflict resolutions are not preserved.
  (you should use `git rebase` only when the history is linear)

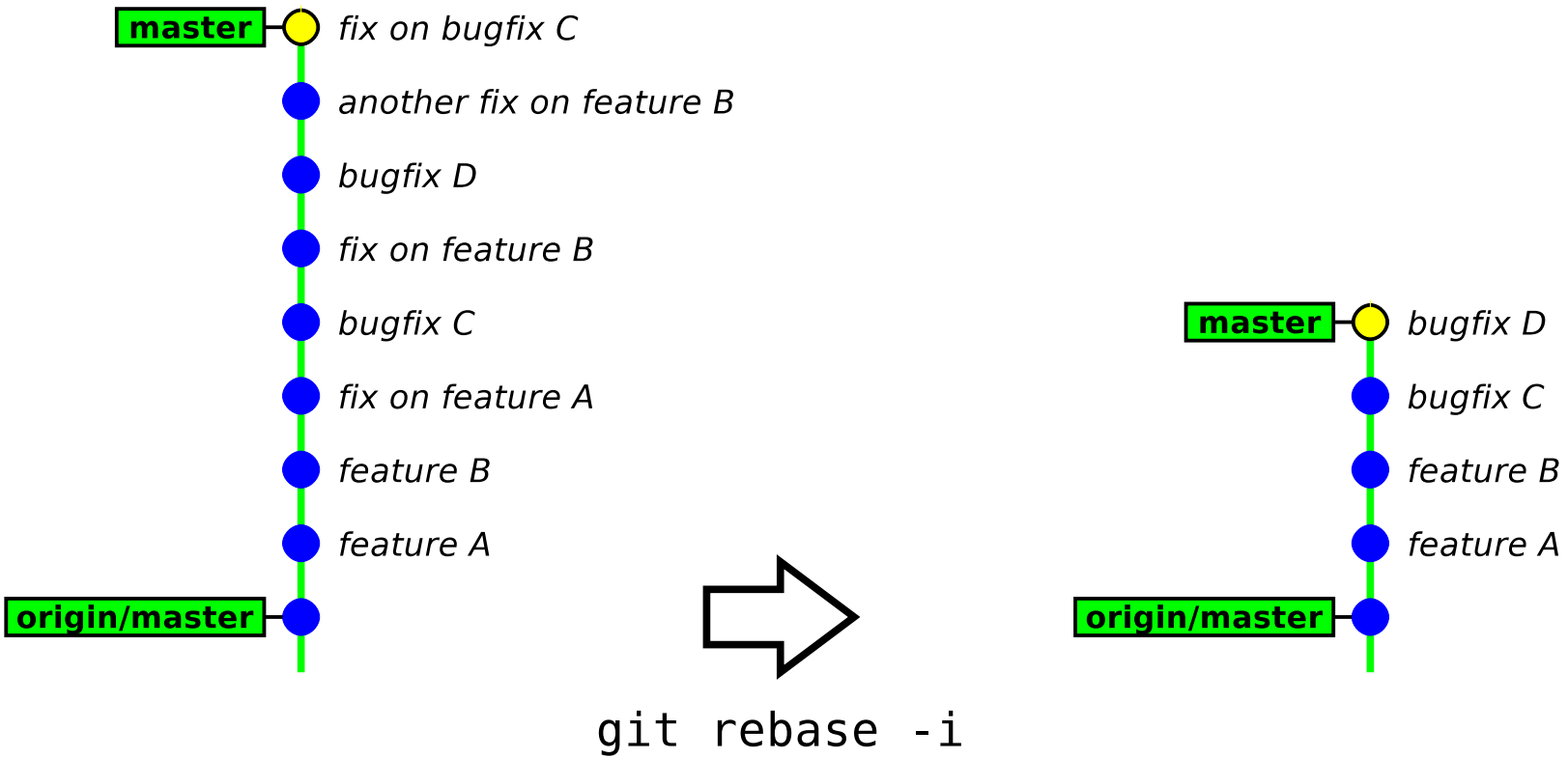
- Conflicts are resolved on a per-commit basis. If both branches updated the same content multiple times, then you will have to resolve the conflicts multiple times.
  (you should prefer using `git merge` instead)

Rebasing onto remote branches

```
git pull --rebase
```

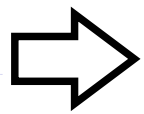


Cleaning your history



Interactive rebase

```
pick b33c7ba feature A
pick f93691c feature B
pick 7b5e038 fix on feature A
pick 540b05b bugfix C
pick c2d6bce fix on feature B
pick ad07f1c bugfix D
pick 90f9473 another fix on feature B
pick 87c1e1d fix on bugfix C

# Rebase b140790..87c1e1d onto b140790
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the
#  e, edit = use commit, but stop for an
#  s, squash = use commit, but meld into
#  f, fixup = like "squash", but discarc
#  x, exec = run command (the rest of th
#
# These lines can be re-ordered; they ar
#
# If you remove a line here THAT COMMIT
# However, if you remove everything, the
#
~
~
~
```



```
pick b33c7ba feature A
f 7b5e038 fix on feature A

pick f93691c feature B
f c2d6bce fix on feature B
f 90f9473 another fix on feature B

pick 540b05b bugfix C
f 87c1e1d fix on bugfix C

pick ad07f1c bugfix D

# Rebase b140790..87c1e1d onto b140790
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the
#  e, edit = use commit, but stop for an
#  s, squash = use commit, but meld into
#  f, fixup = like "squash", but discarc
#  x, exec = run command (the rest of th
#
# These lines can be re-ordered; they ar
#
# If you remove a line here THAT COMMIT
# However, if you remove everything, the
#
```

Split a commit into multiple commits

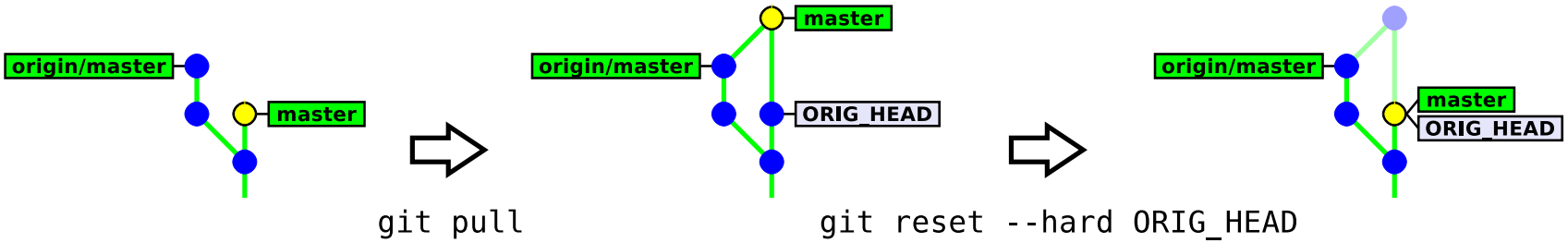
Split the last commit:

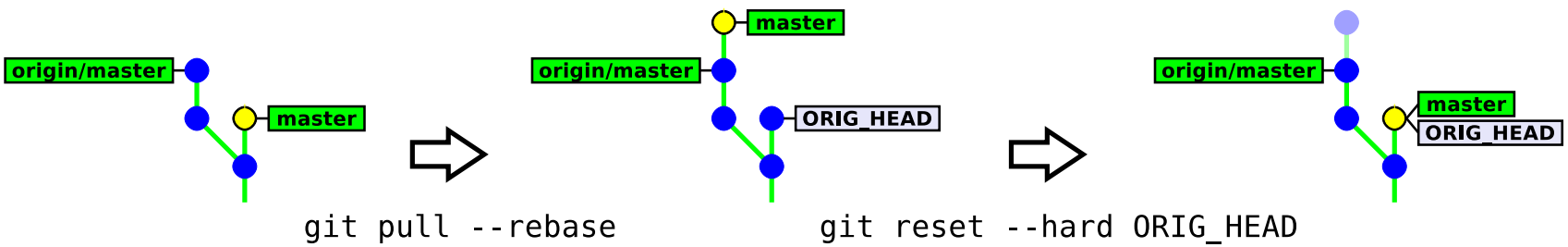
- amend the commit (with `git gui` for example) to unstage parts from the current commit and make additional commits

Split an arbitrary commit:

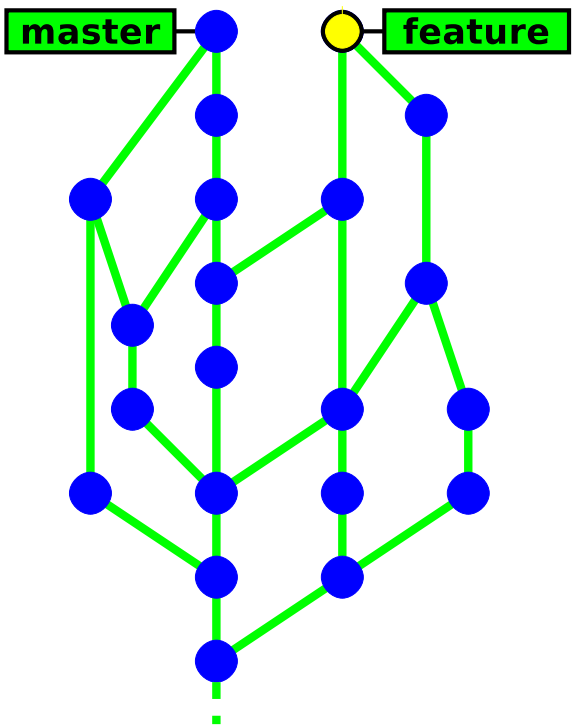
1. run `git rebase -i` and mark the target commit with *edit*
2. git pauses after having applied this commit
   (the current commit is the target commit $\rightarrow$ amend it)
3. run `git rebase --continue` to finish rebasing

Rollback after merge/rebase

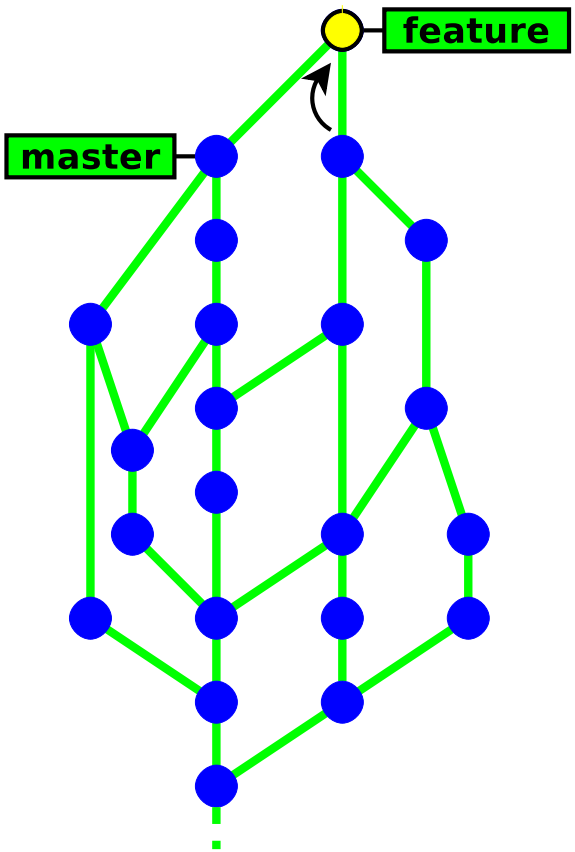




Rewriting a long-lived branch

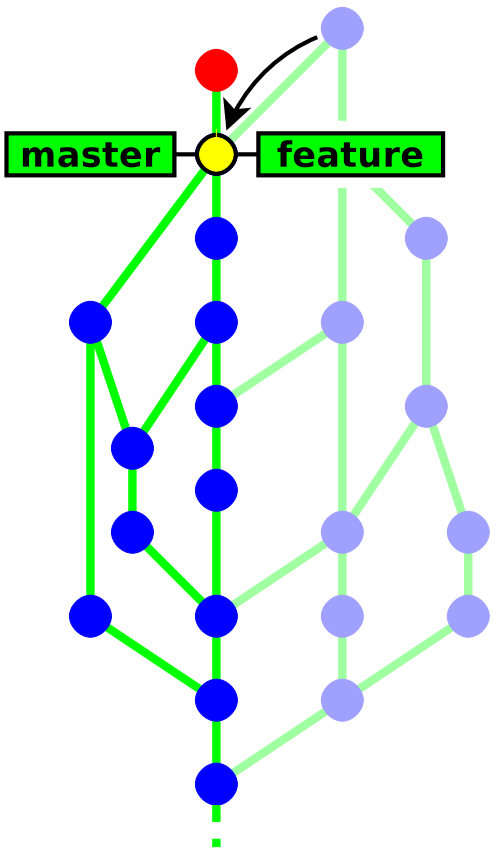


Rewriting a long-lived branch



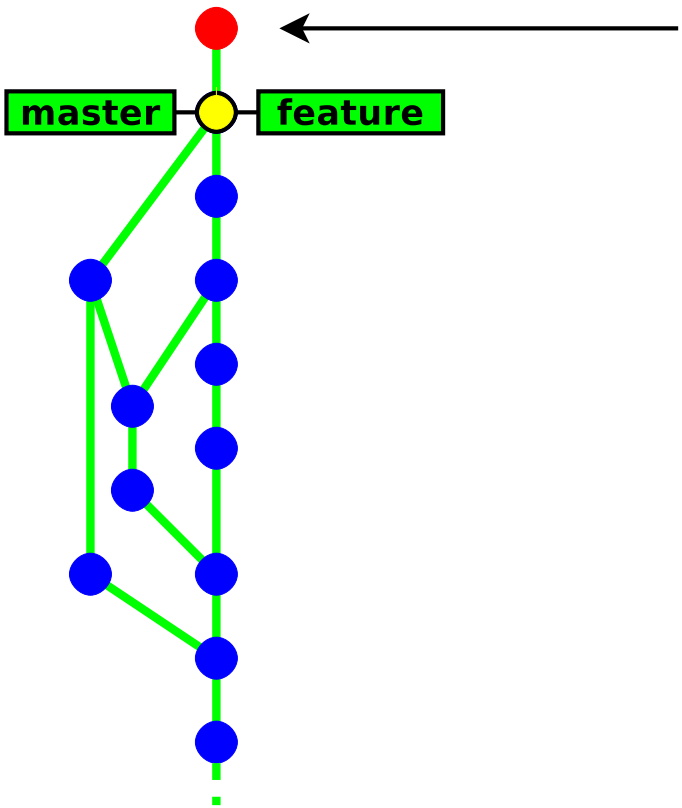

```
git merge master
```

Rewriting a long-lived branch




```
git reset master
```

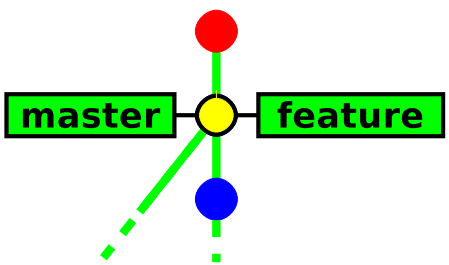
Rewriting a long-lived branch



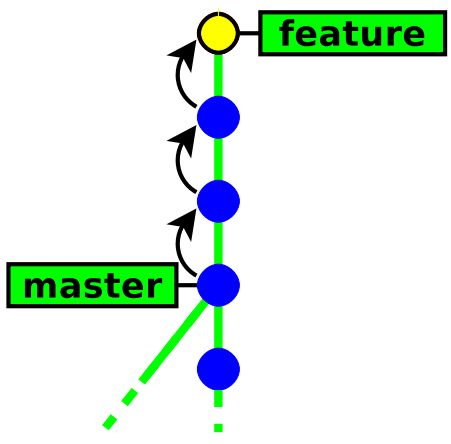
Note:

our changes are still present in the working copy

Rewriting a long-lived branch



Rewriting a long-lived branch



remake the commits
(see part 4: staging hunks/lines)

Rewriting the whole history with `git filter-repo`

https://github.com/newren/git-filter-repo

Possible applications:

- remove confidential content
- fix author names
- extract a subdirectory
- ...

Example: removing a confidential file (`passwords.txt`)

```
$ git filter-repo --invert-paths --path passwords.txt
```

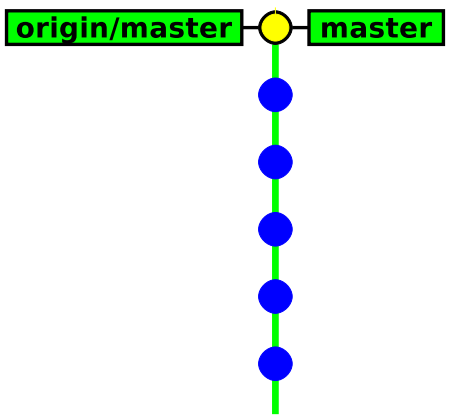
Note:

- ⚠ very destructive command, try it on a temporary clone first
- git also has a native command for rewriting history (`git filter-branch`) but its use is officially discouraged

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

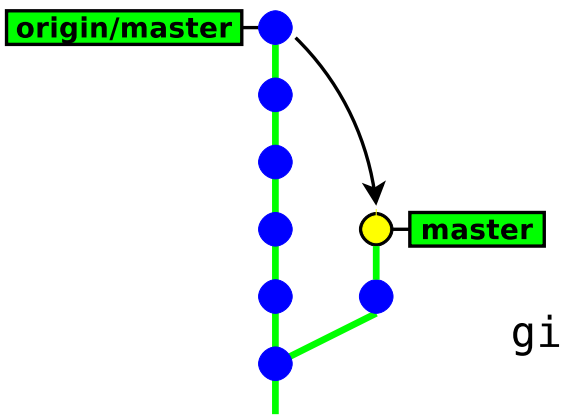


[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless



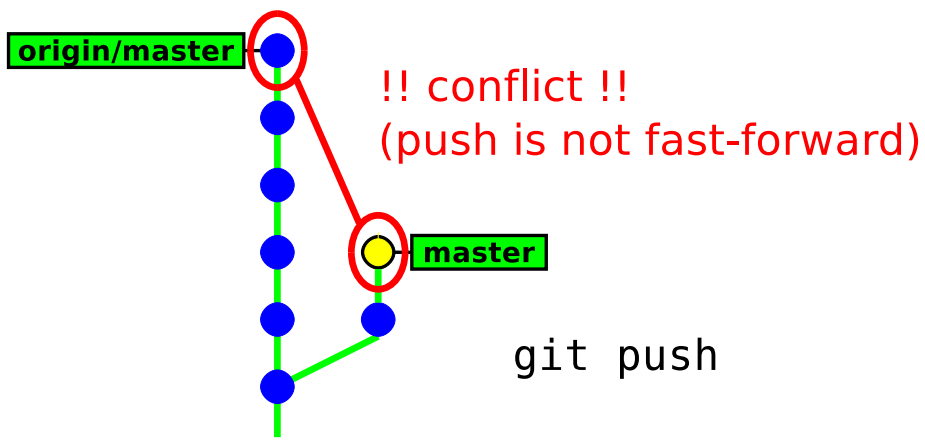

```
git rebase -i HEAD~5
```

[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

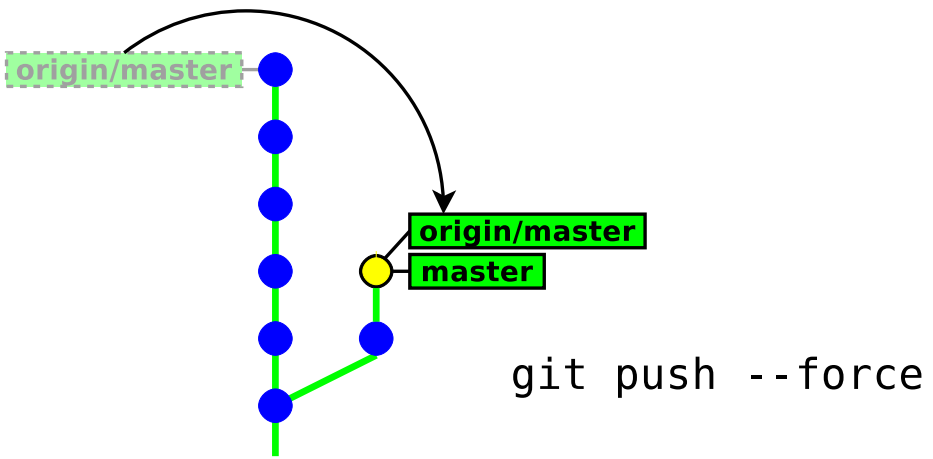


[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

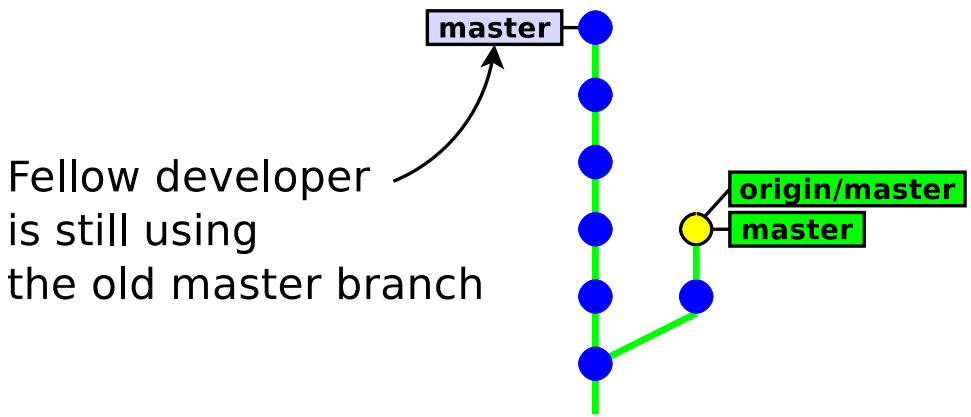


[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless



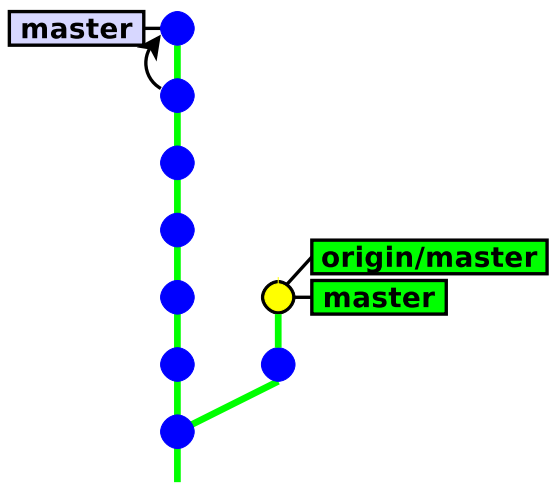
[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

`git commit`



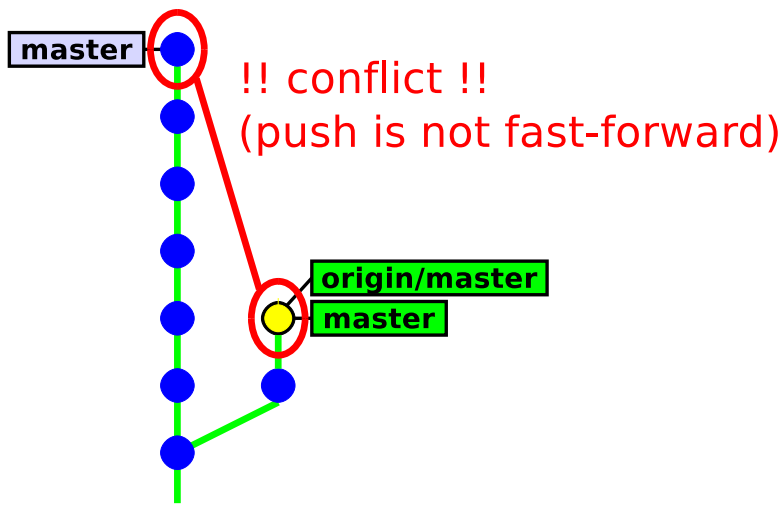
[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

`git push`



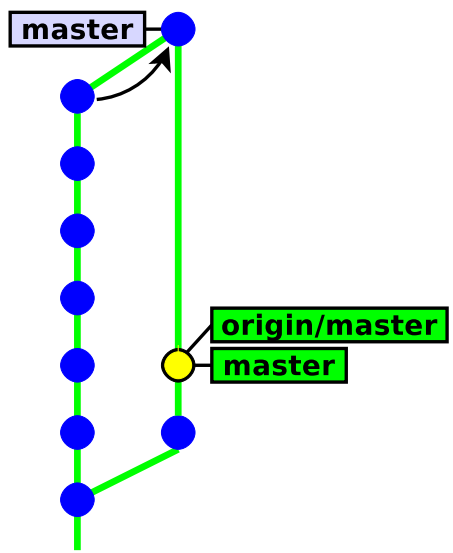
[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

`git pull`



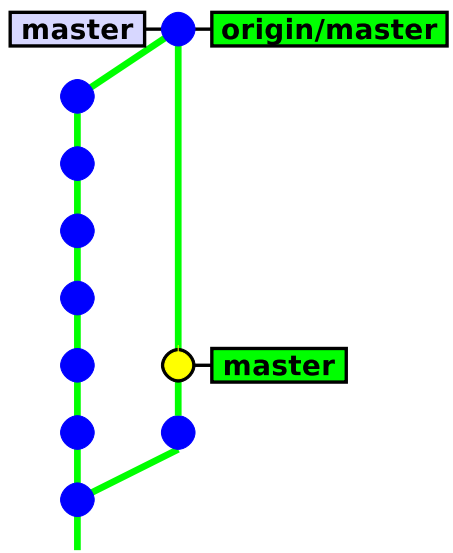
[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

```
git push
```

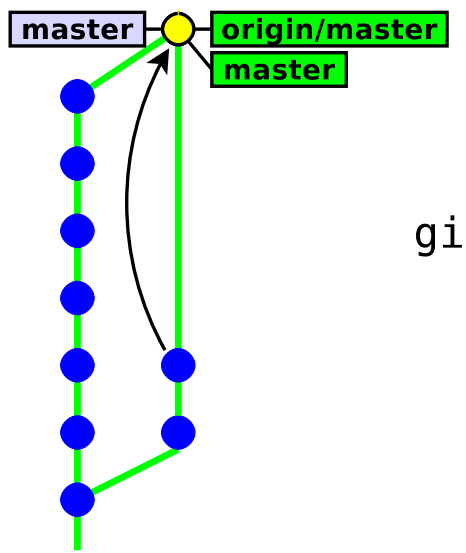


[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless



`git pull`

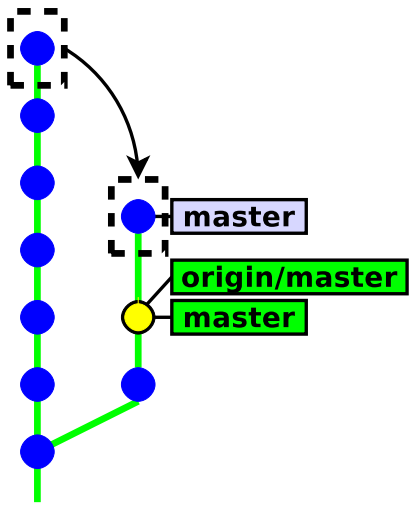
[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

With great power comes great responsibility

Before playing with your history, be aware that:

- the patch's history will be erased forever
- rewriting published branches is disruptive and mostly useless

```
git rebase HEAD~1
   --onto origin/master
```



$\rightarrow$ avoid rewriting branches that were already published[5]

[5] See the bottom of `http://justinhileman.info/article/git-pretty/`

Part 6.

Interoperating with other version control systems

Strategies for using foreign repositories with git

- uni-directional (import)
  `svn-all-fast-export`
  `hg-fast-export`

- bi-directional (live interaction)
  - client-side translation
    `git-svn`
    `git-remote-hg`

  - server-side translation
    `subgit`
    `git-svnsync`

`git-svn`: interaction with a svn server

Purpose : use git locally and synchronise with a remote svn repository

$\rightarrow$ allows offline development

Cloning a SVN repository:

`git svn clone` *svnurl* [*directory*]

```
$ git svn clone svn+ssh://some.server.com/svn/myproject/trunk myproject
```

SVN sync commands

Pushing commits

```
git svn dcommit
```

Pulling commits

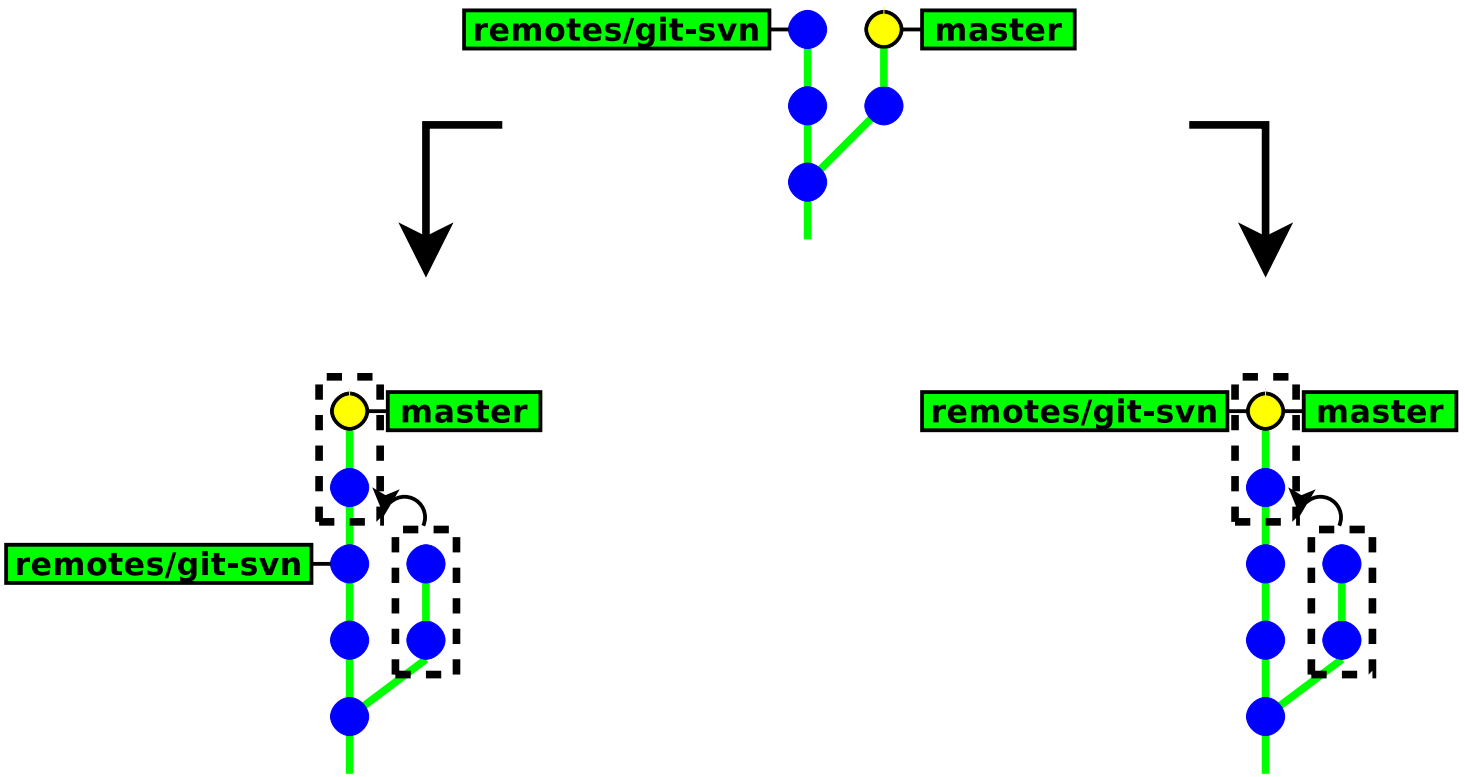
```
git svn rebase
```

Fetching commits

```
git svn fetch
```

`git svn` workflow

`git svn` never merges anything but does a rebase instead[6]



[6] so as to keep a 1:1 mapping between svn and git commits

Cloning a svn repository with multiple branches

```
git svn clone --prefix[7] origin/ --trunk=trunk_path
    --branches=branches_path --tags=tags_path url [ dir ]
git svn clone --prefix[7] origin/ --stdlayout url [ dir ]
```

trunk_path, *branches_path* and *tags_path* are location of the trunk, branches and tags directory relative to the *svnurl*.

`--stdlayout` is an alias for `--trunk=trunk --branches=branches --tags=tags`

```
$ git svn clone --trunk=trunk --branches=branches --tags=tags svn+ssh://some.server.com/svn/
myproject

$ git svn clone --stdlayout svn+ssh://some.server.com/svn/myproject
```

[7] With git<2.0, setting a prefix for remote branches is highy recommended (otherwise it behaves badly). Later versions use `origin/` as default prefix.

`git-svn` shortcomings

`git-svn` is an early hack, it does not fit with git's branching model

- `git svn` commands (`fetch`, `rebase`, `dcommit`) do not preserve the branching history
  → *avoid using* `git merge` *on svn branches*

- svn servers are desperately slow
  → *import only the recent history (use shallow clones)*

git-svn recipes

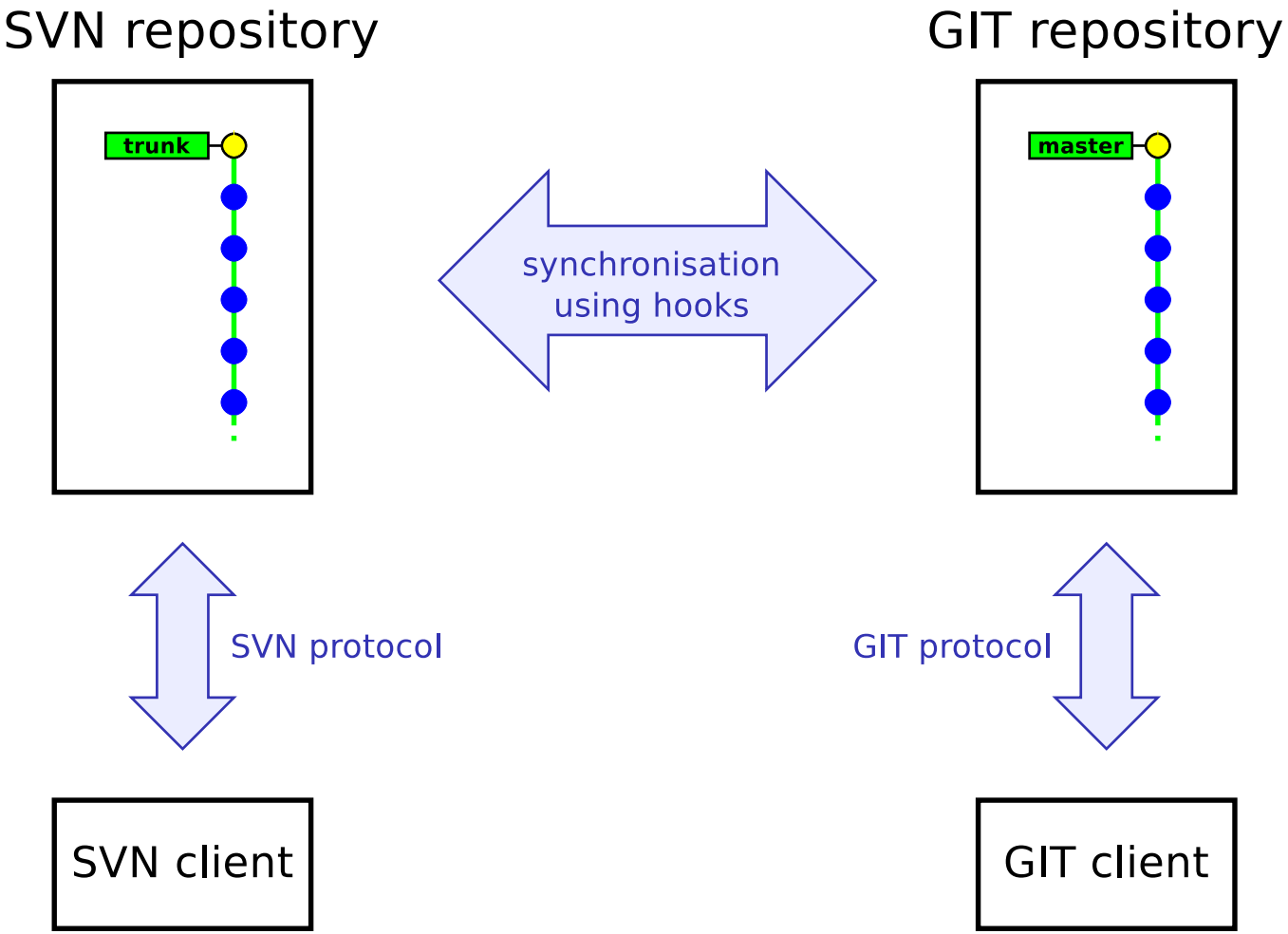
Create a new svn branch

```
git svn branch branch [ start_point ]
```

Shallow clone (fetch only revisions younger than *n*)

```
git svn clone -r n:HEAD svnurl
```

Server-side synchronisation (subgit, git-svnsync)



Interoperating with a mercurial repository

- not yet in the official distribution
  → download the script and install it in your PATH

  `https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg`

- when cloning from mercurial prepend `hg::` before the url

```
$ git clone hg::http://hg.python.org/cpython
```

- use it like any git remote repository

Differences between hg branches and git branches

Mercurial supports different kinds of branches[8]:

- **named branches** which are permanent *(like svn branches)*
- **bookmarks** which are mutable *(like git branches)*

Named branches are mapped in the "`branches/`" namespace (except the `default` branch which is mapped as `master`)

HG branch	GIT branch
`default`	`master`
branch_name	`branches/`*branch_name*
bookmark_name	*bookmark_name*

[8] `http://stevelosh.com/blog/2009/08/a-guide-to-branching-in-mercurial/`

Named branch oddity

- In mercurial there can be multiple named branches with the same name[9] *(they are sometimes referred as anonymous branches)*

  In case of conflict between two tips with the same name, mercurial selects the most recent one.

- Git is unable to represent multiple remote branches with the same name, you may have troubles when someone does `hg push --force`

[9] actually a "named branch" is just an attribute in mercurial commits

Part 7.
Submodules and Subtrees

Use cases

Submodules and subtrees handle dependencies between software stored in separate repositories

Examples:

- your software is closely dependent to an unstable library which evolves rapidly
- part of your software is released to the public, yet you want to keep it synced with the private repository
- you are developping a “informal” library, it is used in multiple projects but you do not want to package it
- your software depends on a library that you need to patch heavily

Basics

The `submodule` and `subtree` map the content of a foreign repository (the subproject) into a subdirectory of your repository.

Example:

- the application *myapp* depends closely on the lib *mylib*
- $\rightarrow$ in the *myapp* repository, you will create a **submodule** (or **subtree**) which embeds the *mylib* repository

Submodules vs Subtree

Submodules and subtrees are similar, except that:

- `submodule` stores just a reference (sha1) of the current commit of the subproject repository
- `subtree` makes a verbatim copy of the subproject repository

`git submodule` is an early hack. It has many issues
→ when possible, use `git subtree` instead

Submodules

Create a submodule

```
git submodule add url path
git commit
```

→ will map the repository *url* into the subdirectory *path*

```
$ git submodule add git://bar/subproject.git/ subproject
Cloning into 'subproject'...
done.
$ git commit -m "new submodule"
[master (root-commit) 492b4a8] new submodule
2 files changed, 4 insertions(+)
$ git ls-tree HEAD
100644 blob caec8b355889c199bdb34ee52aaebecb1e09bc8a    .gitmodules
160000 commit de83a6616386012abbbf27f6b83d6312e3153f03  subproject
$ git remote -v
origin  git://foo/project.git (fetch)
origin  git://foo/project.git (push)
$ cd subproject
$ git remote -v
origin  git://bar/subproject.git (fetch)
origin  git://bar/subproject.git (push)
```

Pulling upstream changes on a submodule

`git pull` does not touch the child repository

→ you must to update it separately

```
git submodule update [ --init --recursive ]
```

```
$ git pull
   c9605b7..6bea4a5  master     -> origin/master
 subproject |    2 +-
$ git status
# On branch master
# Changes not staged for commit:
#
#       modified:   subproject (new commits)
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git submodule update
Submodule path 'subproject': checked out '3bd17b0ad38a93f1da82c3d27f852e67be9af705'
$ git status
# On branch master
nothing to commit (working directory clean)
```

Committing changes inside the submodule

1. Make sure the submodule is **not in detached HEAD** state.
   If it is, then you should run (inside the submodule):
   `git merge master && git checkout -B master`

2. Make your changes in the submodule and `commit` them

3. Push your changes in the submodule remote repository

4. Go back to the parent directory (in the parent repository)

5. `commit` the submodule (to update the reference to the submodule)

6. Push your commit

Limits of submodules

- submodules tracks commits, not branches or tags
  → `submodule update` leaves the module in detached HEAD
- concurrent submodules updates cannot be merged easily (just like merging binary files)
- `git pull` does not update submodules automatically
  → previous commits may be accidentally reverted when committing before updating a submodule
- using submodules requires all developers to know about submodules (at least `git submodule update`)
- lots of commands to be run, even for simple operations, any mistake may break something badly

Subtree

`git subtree` is a recent addition (still in the contribs)
→ download the script[10] and install it in your PATH

`https://raw.github.com/git/git/master/contrib/subtree/git-subtree.sh`

`git subtree` adderess similar use cases as `git submodule` but in a more distributed fashion

- the subtree is cloned inside the parent project
- commits done in the subtree do not need to be propagated upstream immediately

[10] you may want the man page too: `https://raw.github.../git-subtree.txt`

`subtree` in comparison with `submodule`

- Advantages
  - commands are simpler and safer
  - only the developer doing the sync (push/pull) with the foreign repository need to understand the `subtree` command
    - no special actions needed after `git clone`
    - making commits in a subtree is straigthforward
  - subproject code remains available even if the foreign repository is no longer online
- Drawbacks
  - not suitable for large subprojects
  - no cyclic dependencies (infinite recursion)

Creating a subtree

`git subtree add [ --squash ] --prefix` *prefix url ref*

This command imports the commit *ref* (and its history) from the repository *url* and merges it into the current branch in the subdirectory *prefix*.

```
$ git subtree add --prefix subproject git://bar/subproject.git master
git fetch git://bar/subproject.git master
From git://bar/subproject.git
 * branch            master     -> FETCH_HEAD
Added dir 'subproject'
```

With `--squash`, git imports only a single commit from the subproject, rather that its entire history.

Sync with the remote subproject

Receiving changes[11]

```
git subtree pull [ --squash ] --prefix prefix url ref
```

Sending changes

```
git subtree push --prefix prefix url ref
```

Note: you should always do `subtree pull` after each `subtree push` to avoid possible conflicts later on

[11] Note: use the `--squash` option consistently across each invocation

Reviewing changes before push

`git subtree split --prefix` *prefix*

`git subtree split` prepares the commits before pushing to the remote subproject repository. It returns the sha1 of the commit to be pushed.

→ useful for reviewing the commits and generating patches

```
$ git commit -m "first change"
  ...
$ git commit -m "second change"
$ git subtree split -P subproject
982f2b84120e9dab487d2564f0fefff80ce757a9

$ gitk `git subtree split -P subproject`

$ git format-patch  master..`git subtree split -P subproject`
0001-first-change.patch
0002-second-change.patch
```

Alternatives

`submodules` and `subtree` are best used with small leaf projects (without cyclic deps or multiple imports of the same subproject)

Other tools:

- git-repo (works with a collection of git repositories)
  `https://gerrit.googlesource.com/git-repo`
- git-subrepo (similar to git-subtree)
  `https://github.com/ingydotnet/git-subrepo`

Part 8.

Managing a collection of patches

Use case

- You are using a 3rd-party software that you must modify to fit your needs *(features, bugfixes, port, integration with other apps, . . .)*
- You want to merge updates from the upstream project *(possibly conflicting with your local changes)*
- You would like to hand on your patches upstream and decrease maintainance costs

Patches acceptance

There are many reasons for not accepting a patch:

- not compliant with the project policies/coding style
- not mature (quick-and-dirty patch)
- too fragile (may break something, open a security hole)
- too intrusive ($\rightarrow$ difficult to maintain)
- too specific (worthless for the community)

You may need several iterations before the patch is accepted
$\rightarrow$ a patch has its own history

Sorting patches

Patches will have different futures:

- some will be accepted
- some will be accepted with modifications
- some will be refused

$\rightarrow$ you need to sort out your collection of patch as some get accepted

Basic ways of managing patches

- single patch
  - just `commit` in the current branch, and `pull` regularly
  - output the patch: `git diff origin/`*branch*`...`*branch*
  - discard the patch: `git reset --hard origin/`*branch*
- collection of simple patches
  - `commit` in the current branch
  - use `rebase -i` to rearrange the commits (1 commit per patch)
  - use `pull --rebase` to rebase[12] your patches on the upstream code
  - output the patches: `git format-patch origin/`*branch*

[12] if one of your patches is applied upstream, then `rebase` will automatically filter it when rebasing your work

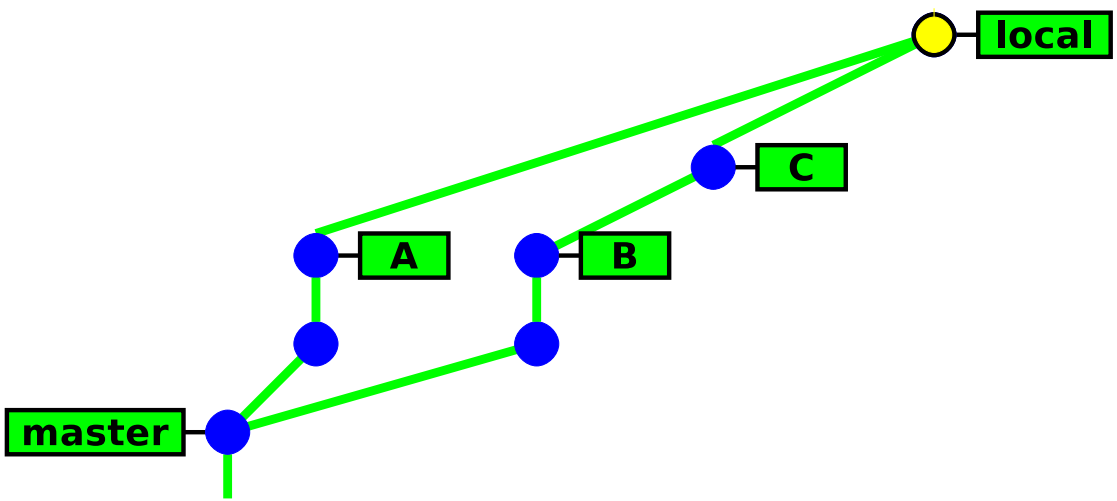
Branch-based patch management

- use one branch per patch
- merge all branches into an integration branch
- rebase the integration branch when discarding a patch
- avoid committing directly in the integration branch

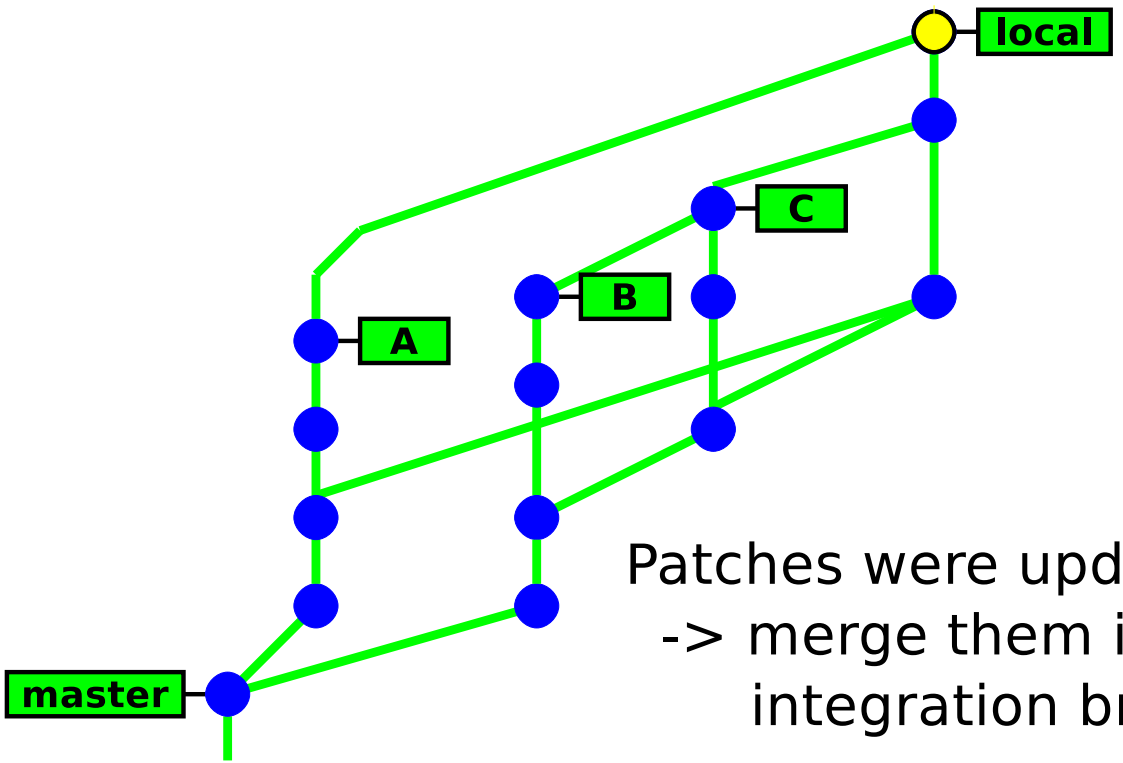
Branch-based example

Three patches: A, B and C
(C depends on B)

-> merge them into an integration branch

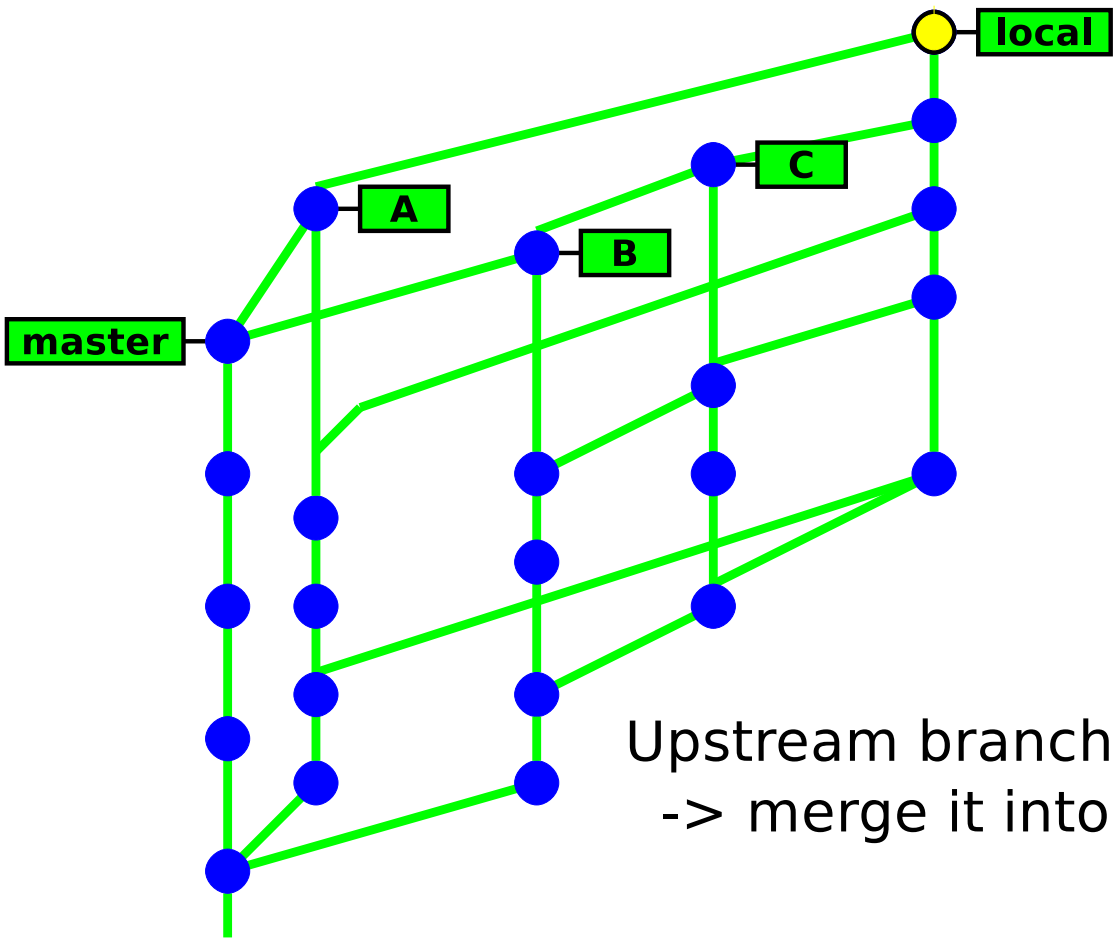


Branch-based example



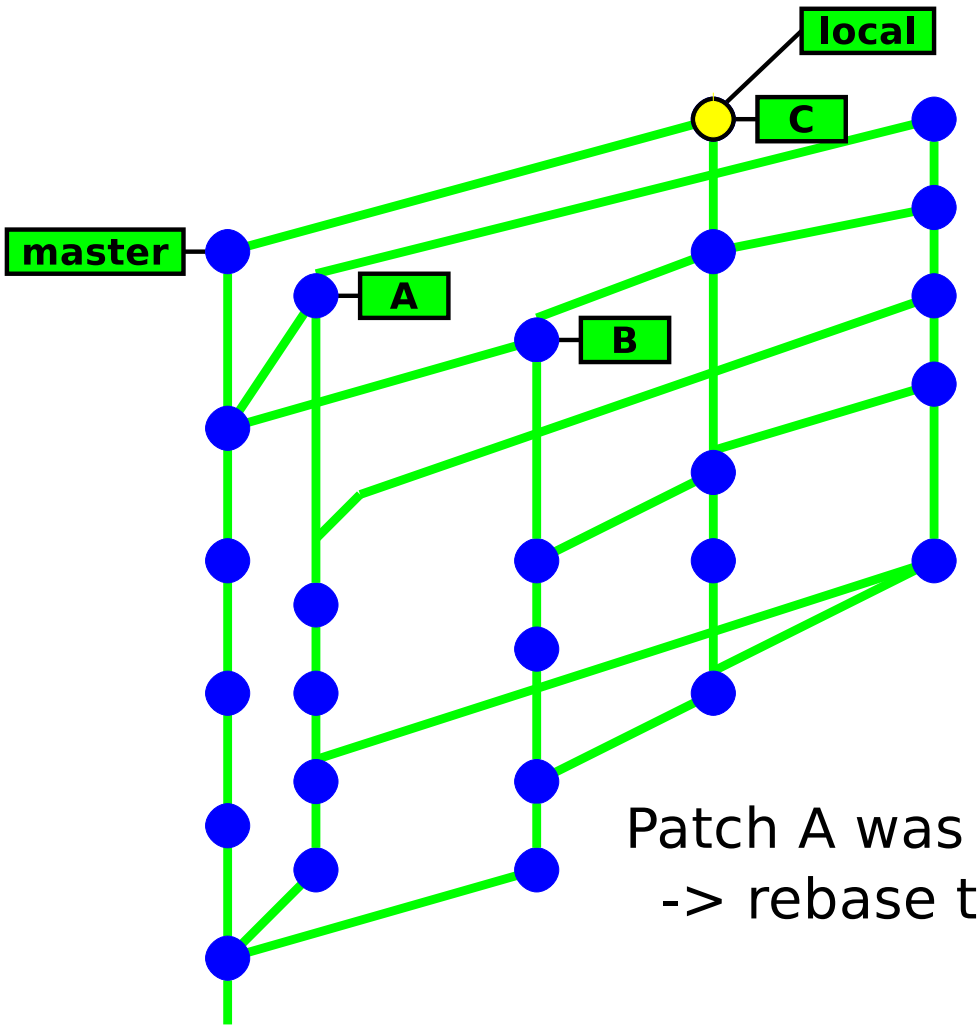
Patches were updated
-> merge them into the
integration branch

Branch-based example



Upstream branch was updated
-> merge it into other branches

Branch-based example



Patch A was applied upstream
-> rebase the integration branch

Tools

There are 3rd party tools to ease managing patches:

- Topgit
- Stacked Git (stgit)
- Guilt
- cj-git-patchtool

Topgit

- branch-based (1 branch per patch)
- metadata stored in the tree (`.topdeps` `.topmsg`)
- pros
  - branches can be shared with other developers
  - workflow similar to git's native workflow
- cons
  - requires switching between branches
  - must care about the metadata when using other git commands (eg. merge)

Topgit main commands

create a patch (new branch)	`tg create` *name* [*deps...*]
merge upstream changes (current branch + deps)	`tg update`
convert commits into topgit branches	`tg import` *range*
convert topgit branches into commits	`tg export` *newbranch*

Stacked Git (stgit)

- stacked-based (1 commit per patch), very similay to quilt
- metadata stored in `.git/patches`
- history saved in a separate branch
- pros
  - good ergonomy
  - usable on multiple branches concurrently
- cons
  - local use only (single user)
  - differences with native git's workflow

Stacked Git main commands

create a patch (new commit)	`stg new` [*name*]
update a patch	`stg refresh` [`-p` *name*] (do not use `git commit`)
move up/down the stack	`stg push` / `stg pop`
import/export patch as diff files	`stg import` [*file/url*] `stg export` [*patches...*]
publish your stack as a branch with a linear history (never rebased)	`stg publish`

Part 9.
Managing large files

The problem with large files

- Git can handle files of any size, but it takes space
- by design the repository contains the content of all files of the current and previous revisions

$\rightarrow$ cloning a repository implies cloning each revision of every file

Solution: store only a hashsum of the file and have the the large files in a special repository

Extensions for handling large files

- `git annex`

  `https://git-annex.branchable.com/`

  - distributed
  - very versatile (many storage backend, many repos)
  - explicit actions (eg: `git annex add`, `git annex copy`)

- `git lfs`

  `https://github.com/git-lfs/git-lfs`

  - centralised
  - very simple specification
  - many implementations
  - mostly transparent to the user
    (uses git's clean/smudge filters and selects files by extension)

Git Annex

- an extension for versionning large files, but without storing their content directly in the repository
- the repository stores only a key made from the hashsum of the content

  eg: `SHA256-s4097177--b7e6cd5fdc215780993cb96e37a8dac735f9307c5d2dfd73c09379e9e16a33ad`
- actual blobs are stored in a separate pool handled by git annex[13] and may be transferred on an on-demand basis

[13] located in `.git/annex/objects`

Technical details

- git annex files are symbolic links pointing to a key somewhere inside `.git/annex/objects`
  - if content is available locally $\rightarrow$ the link is working
  - if content is not available $\rightarrow$ the link is broken
- each repository participating to the annex has a unique id
- objects are indexed in a special branch named `git-annex` and synchronised across all repositories

Getting started

Initialise the annex

```
git annex init [ description ]
```

Add new files into the annex

```
git annex add [ files ... ]
git commit
```

- '`git annex add`' hashes the file, moves it into the annex, replace it with a link pointing to the annex key and stages the link for commit
- '`git commit`' commits the link

$\rightarrow$ the large file is now versionned but not stored in the repository

git annex add example

```
$ git status
Untracked files:
        steal.this.film-vh-prod.avi
$ ls -lh
-rw-r--r-- 1 abaire abaire 351M Jun 10 20:39 steal.this.film-vh-prod.avi

$ git annex add steal.this.film-vh-prod.avi
add steal.this.film-vh-prod.avi ok
(Recording state in git...)

$ git status
Changes to be committed:
        new file:   steal.this.film-vh-prod.avi
$ ls -l
lrwxrwxrwx 1 abaire abaire 202 Jun 10 23:39 steal.this.film-vh-prod.avi -> .git/annex/objects/KX/
xq/SHA256E-s367230976--17c328096c0048c5697135879881f71b8d2eebcc11e6007f15eb0fa52ff35830.avi/
SHA256E-s367230976--17c328096c0048c5697135879881f71b8d2eebcc11e6007f15eb0fa52ff35830.avi

$ git commit -m "nice movie"
[master cb19031] nice movie
1 file changed, 1 insertion(+)
```

Dropping the content of a file

```
git annex drop [ --force ] [ path ... ]
```

Drop the content of *path* from the annex

- if no path is given, then the command applies to all git annex files in the current directory and subdirecties
- unless `--force` is given, git annex will not drop de file unless at least on other copy exists in another repository

git annex drop example

```
$ git annex drop
drop steal.this.film-vh-prod.avi (unsafe)
  Could only verify the existence of 0 out of 1 necessary copies

  Rather than dropping this file, try using: git annex move

  (Use --force to override this check, or adjust numcopies.)
failed
git-annex: drop: 1 failed

$ git annex drop --force
drop steal.this.film-vh-prod.avi ok
(Recording state in git...)

$ cat steal.this.film-vh-prod.avi  >/dev/null
cat: steal.this.film-vh-prod.avi: No such file or directory
```

`git annex` and remote repositories

`git annex` distinguishes two kinds of remote repositories

- native git repositories[14]
- 'special' remotes $\rightarrow$ to store content elsewhere[15]

```
git annex sync [ --content ] [ remote ]
```

$\rightarrow$ Synchronise with a remote repository:

- propagate the annex index across repositories
- with `--content`, the objects are propagated too

[14] only if accessed with ssh and if git-annex-shell is available there

[15] current list of drivers include : S3, rsync, webdav, web, bup, bittorrent, ...

$\rightarrow$ `https://git-annex.branchable.com/special_remotes/`

git annex sync example

```
$ git annex whereis
whereis steal.this.film-vh-prod.avi (1 copy)
        2bced7dc-b9d6-4ab5-9d1e-ec56e3a8fe36 -- [here]
ok

$ git annex sync --content
[...]
From /tmp/hello
 * [new branch]      git-annex  -> origin/git-annex
(merging origin/git-annex into git-annex...)
(Recording state in git...)
copy steal.this.film-vh-prod.avi copy steal.this.film-vh-prod.avi (to origin...) ok
To /tmp/hello.git
 * [new branch]      git-annex -> synced/git-annex
 * [new branch]      master -> synced/master
ok

$ git annex whereis
whereis steal.this.film-vh-prod.avi (2 copies)
        168a147e-f80e-4be0-869d-56526b208527 -- [origin]
        2bced7dc-b9d6-4ab5-9d1e-ec56e3a8fe36 -- [here]
ok

$ git annex drop steal.this.film-vh-prod.avi
drop steal.this.film-vh-prod.avi ok
```

`sync` caveats

`git annex sync` is a kind of 'magic' command (`git annex` was designed as a distributed backup tool)

- ⚠ `sync` automatically commits all changes in the working copy before syncing
- ⚠ by default `sync` works **with all remote repositories**. If you do not want this behaviour, then you must give explicit remote names.
- conflicts on git annex files are handled automatically
  → will store both files in the result
  (eg: `foo.somekey` and `foo.otherkey`)
- `--content` syncs all files bidirectionnally, if you want unidirectionnal sync only, you may prefer '`git annex copy --to` *remote*' instead

Other useful commands

copy/move files from/to a remote repository

```
git annex copy --to/--from remote [ path ]
git annex move --to/--from remote [ path ]
```

download a files from any remote repository

```
git annex get [ path ]
```

display/drop unused keys[16]

```
git annex unused
git annex dropunused NUM|RANGE
```

[16] ⚠ 'unused' means 'not referred by any file in the **current revision**' files from older revisions may be reported as unused if not present in the current revision

Part 10. Scaling

The linux case

A selection of interesting articles:

- *Why Github can't host the Linux Kernel Community*
  `https://blog.ffwll.ch/2017/08/github-why-cant-host-the-kernel.html`
- *Submitting patches: the essential guide to getting your code into the kernel*
  `https://www.kernel.org/doc/html/latest/process/submitting-patches.html`
- *Rebasing and merging: some git best practices*
  `https://lwn.net/Articles/328436/`
- *What's missing from our changelogs*
  `https://lwn.net/Articles/560392/`