PERFORCE
Version everything.

Dive into git

Jan Van Uytven

```
GIT(1)                          Git Manual                          GIT(1)



NAME
       git - the stupid content tracker

SYNOPSIS
       git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

PERFORCE
Version everything.

Outline

- Git Overview
- Git Architecture
- Git Common Tasks
- Git Remotes
- Further Topics
- Q & A

PERFORCE
Version everything.

A Brief History of Git

- Up until 2005, The Linux Project was hosted using the proprietary BitKeeper DVCS.
- In March, Larry McVoy (BitKeeper author) revoked the free BitKeeper license granted to the Linux Team, accusing Andrew Tridgell (a kernel developer) of reverse-engineering the BitKeeper protocols.
- Linus Torvalds & co began evaluating open-source alternatives (SVN, Monotone)

A Brief History of Git

PERFORCE
Version everything.

- Linus Torvalds & co had a list of requirements for the new SCM.
- None of the existing open-source SCMs fulfilled the requirements.
- Linux realized he would need to create one from scratch.

Requirements:

1. Had to be a DVCS, like BitKeeper
2. Handle thousands of developers
3. Fast and efficient, with a clean object model
4. Maintain internal integrity
5. Enforce accountability
6. Immutable objects, but not immutable history
7. Atomic transactions
8. Branching needs to be fast and easy
9. A repositories should be complete.
10. Free, as in beer.

A Brief History of Git

PERFORCE
Version everything.

- Development began on April 3rd
- Git became self-hosting on April 7th

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af290
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700

    Initial revision of "git", the information manager from hell
```

A Brief History of Git

PERFORCE
Version everything.

SHARE

COLLABORATE

- Development began on April 3rd
- Git became self-hosting on April 7th

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af290
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700

    Initial revision of "git", the information manager from hell
```

- First commit of Linux kernel into git happened on April 16th

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Sat Apr 16 15:20:36 2005 -0700

    Linux-2.6.12-rc2

    Initial git repository build. I'm not bothering with the full history,
    even though we have it. We can create a separate "historical" git
    archive of that later if we want to, and in the meantime it's about
    3.2GB when imported into git - space that would just make the early
    git days unnecessarily complicated, when we don't have a lot of good
    infrastructure for it.

    Let it rip!
```

A Brief History of Git

- Development began on April 3rd
- Git became self-hosting on April 7th

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af290
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700

    Initial revision of "git", the information manager from hell
```

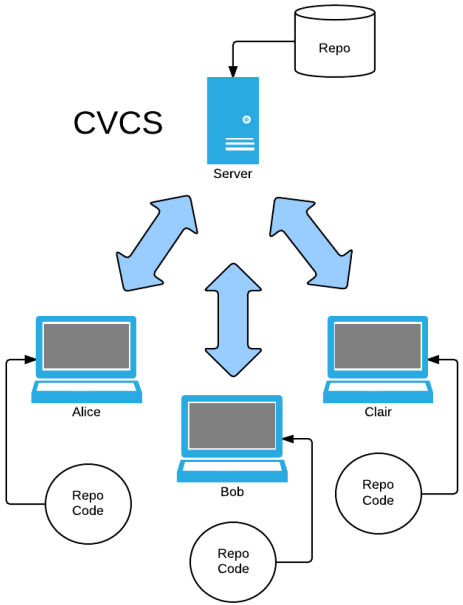
- First commit of Linux kernel into git happened on April 16th

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Sat Apr 16 15:20:36 2005 -0700

    Linux-2.6.12-rc2

    Initial git repository build. I'm not bothering with the full history,
    even though we have it. We can create a separate "historical" git
    archive of that later if we want to, and in the meantime it's about
    3.2GB when imported into git - space that would just make the early
    git days unnecessarily complicated, when we don't have a lot of good
    infrastructure for it.

    Let it rip!
```

- < 2 weeks from start to finish

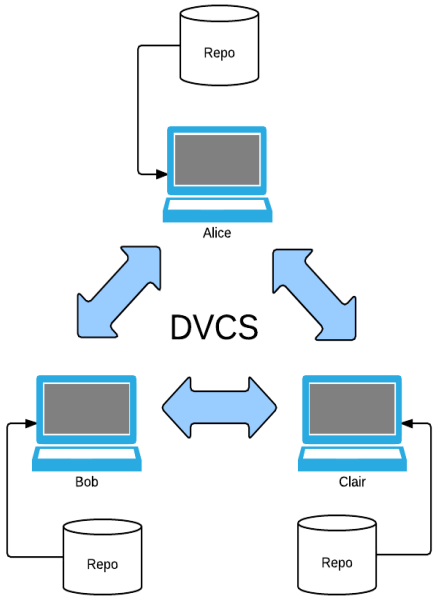
What is a DVCS?

PERFORCE
Version everything.



- In a centralized version control system, the change history is stored in a central location, and everyone retrieves and contributes code to and from this one source.
- People can have the whole codebase, or a portion of it.
- The distinction between client (the developer) and server (the repo) is clear.

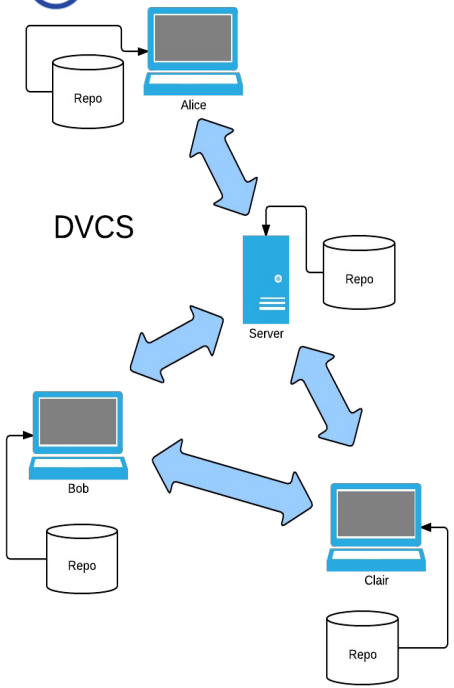
What is a DVCS?



- With a distributed version control system, everyone gets a copy of the repo, and changes can be propagated from any user to any other user, subject to any security constraints.
- Each user acts as an ad-hoc server, therefore, no centralized server is required.
- However...

What is a DVCS?

PERFORCE
Version everything.



- In practice, there is almost always a 'central' or 'authoritative' server.
- However this is through consensus, and the 'central' server has no special attributes.
- Any repo can be declared as the new 'authoritative' source as needed.
- Change can still flow between other repos if desired.

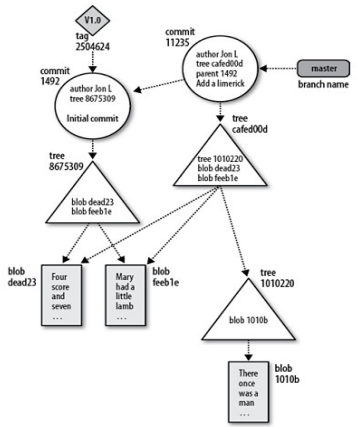
What is a DVCS?

PERFORCE
Version everything.

- DVCSs have exploded in popularity recently. Why?
  - Disconnected operation – work anywhere, anytime
  - Experimental branching – creating and destroying branches is easy
  - Ad-hoc collaboration with peers
  - DVCS stays out of the way

PERFORCE
Version everything.

Git Architecture



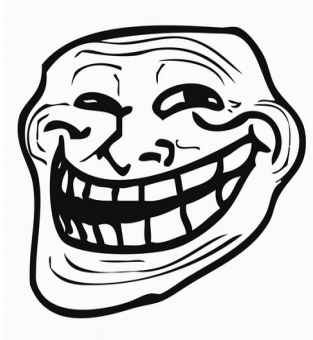
Let's Talk About Hashes

Let's Talk About Hashes

PERFORCE
Version everything.

- Git uses the SHA-1 hash to identify almost everything it stores
  - SHA-1 is a hashing algorithm that produces the same 40-character hexadecimal string for any distinct piece of content
  - Chance of collision: one in 2^80 blobs
  - Example:

```
$ git hash-object troll.png
  ac9b69eebf06207e8478f5a308364cb6cb98899a
```



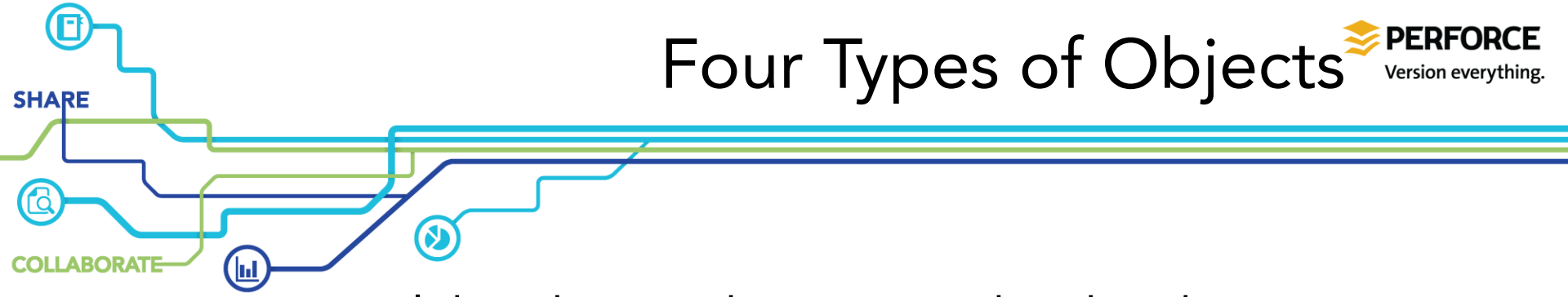
Let's Talk About Hashes

PERFORCE
Version everything.

- In Git, it is not necessary to specify the full hash, as long as Git can *unambiguously* identify what you mean.
- The minimum length is 4 characters.
- You can use `git rev-parse` to figure out the full hash, if needed.

For example, the following are all equivalent:

```
$ git show ed0e6e213cdf11330c194530f80d19545dab4842cv
$ git show ed0e6e213
$ git show ed0e
```

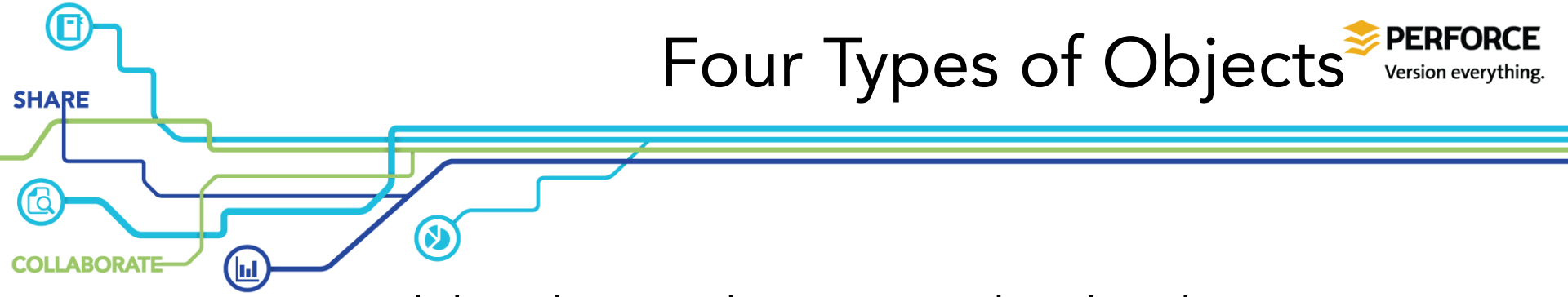


Four Types of Objects

PERFORCE
Version everything.

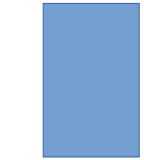
```
blob
e4f6
```

- Blobs – binary objects stored in the object store. They can contain other git objects (trees, commits, tags) and the content of files.

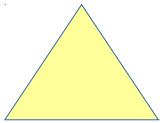


Four Types of Objects

PERFORCE
Version everything.



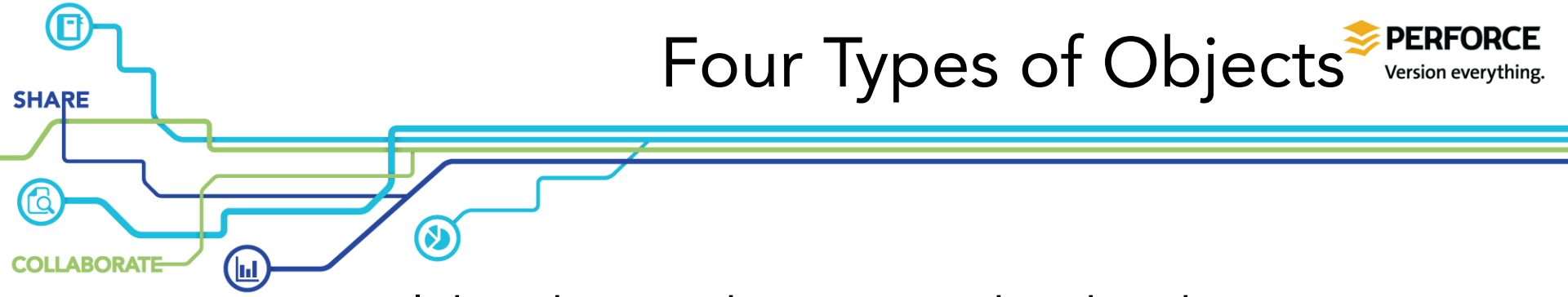
`blob`
`e4f6`

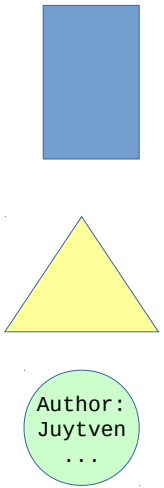


`tree`
`ca5d`

- Blobs – binary objects stored in the object store. They can contain other git objects (trees, commits, tags) and the content of files
- Trees – simple structures that represent a single directory layer

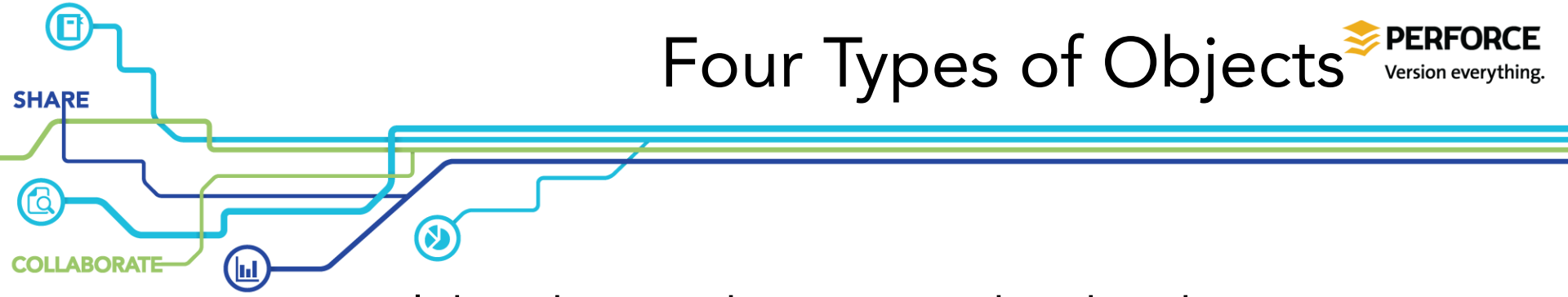
Four Types of Objects



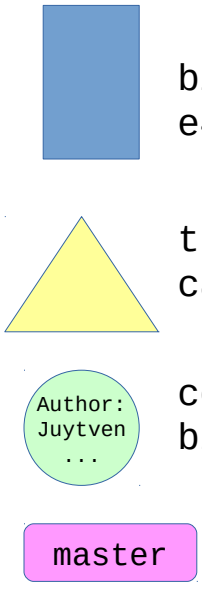


- Blobs – binary objects stored in the object store. They can contain other git objects (trees, commits, tags) and the content of files
- Trees – simple structures that represent a single directory layer
- Commits – points to a single tree and potentially one or more parent commits

Four Types of Objects



PERFORCE
Version everything.



- Blobs – binary objects stored in the object store. They can contain other git objects (trees, commits, tags) and the content of files
- Trees – simple structures that represent a single directory layer
- Commits – points to a single tree and potentially one or more parent commits
- Refs – tags that point to a particular commit, or to other refs (the latter are called *symbolic refs*)

Blobs

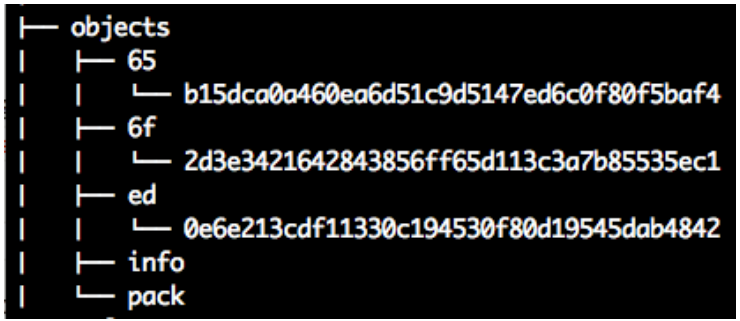
PERFORCE
Version everything.

- All repository data, excluding the index, references and a few special pieces of metadata, are stored as blobs in git's *object store* (.git/objects)
- The blobs are named by the SHA-1 hash of their content.*
- The object store has a flat namespace, broken up by the first two characters of the object's SHA-1 hash. For example, the filename `64c0dbc6305b775c3031f6ba221303c36416dfba` is stored as `64/c0dbc6305b775c3031f6ba221303c36416dfba`
- This divides the object store into 256 directories, which helps avoid filesystem problems and helps the performance of calls such as readdir().

* Well, their content prepended with 'blob <n>' where n is the file length. This is done so git can store 0-byte entities, like empty files.

Blobs

PERFORCE
Version everything.



- You can examine the contents of any blob with `git cat-file -p <sha1>`
- Blobs do not contain any metadata or structure – only the contents of the object they store. Not even the filename!

- Because every object with the same contents will hash to the same blob, it's possible to aggregate object stores across repositories.

Trees

PERFORCE
Version everything.

- If Git blobs don't know their own names, how are they tracked?
- If Git blobs are stored in a flat namespace, how is directory structure stored?

Trees

PERFORCE
Version everything.

- Git uses tree objects to track a particular directory level.
- Each tree lists the files (and blob hashes) in that particular directory, and points to additional trees for each sub-directory.
- You can use git ls-tree <tree> to list the contents of a particular tree:

Trees

PERFORCE
Version everything.

- Git uses tree objects to track a particular directory level.
- Each tree lists the blobs in that particular directory, and points to additional trees for each sub-directory.
- You can use git ls-tree <tree> to list the contents of a particular tree:

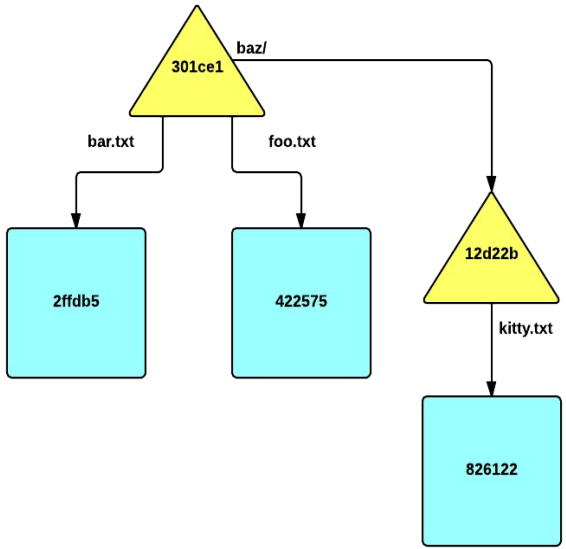
foo.txt
bar.txt
baz/kitty.txt

Trees

PERFORCE
Version everything.

- Git uses tree objects to track a particular directory level.
- Each tree lists the blobs in that particular directory, and points to additional trees for each sub-directory.
- You can use git ls-tree <tree> to list the contents of a particular tree:

foo.txt
bar.txt
baz/kitty.txt



Trees

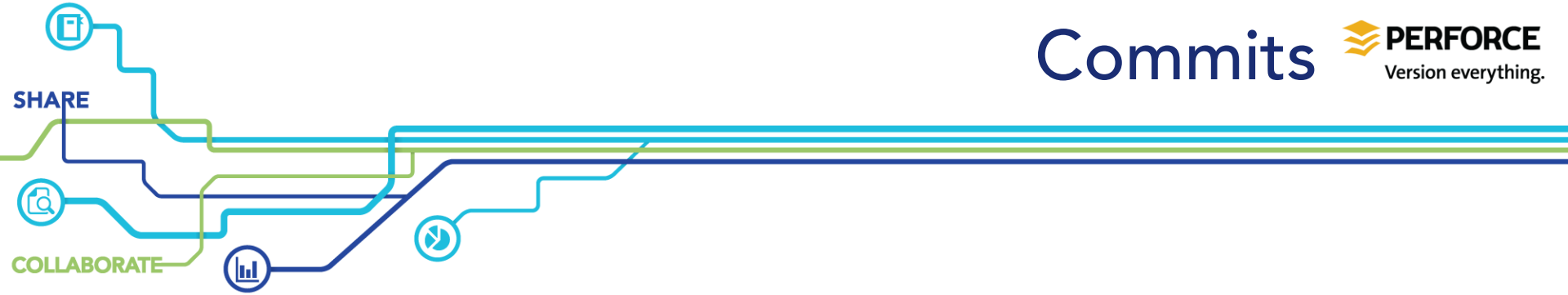
PERFORCE
Version everything.

- Git uses tree objects to track a particular directory level.
- Each tree lists the blobs in that particular directory, and points to additional trees for each sub-directory.
- You can use git ls-tree <tree> to list the contents of a particular tree:

foo.txt
bar.txt
baz/kitty.txt

```
➜  simple-2 git:(master) git ls-tree master
100644 blob 2ffdb5cc9a74efa023c0ada3238a198f59032a90    bar.txt
040000 tree 1d2d2b3771930d5d92ee985b693a8d036c770b5e    baz
100644 blob 4225756afde305d57da8f324025b6c6d286eb789    foo.txt
➜  simple-2 git:(master) git ls-tree 1d2d2b
100644 blob 826122389de1ecabb671b1e7e7bb52869e720a47    kitty.txt
➜  simple-2 git:(master)
```

This works because ls-tree takes a 'tree-ish' object. It's smart enough to realize that master points to a commit, and the commit, in turn, contains a link to the root tree object for that commit



Commits

PERFORCE
Version everything.

- So blobs contain the contents of files stored by Git
- Trees contain the information to reconstitute a single directory and all the files that directory contains, as well as links to other trees representing subdirectories.
- Thus given a root tree(a tree corresponding to the root of the repo) we can retrieve an entire snapshot of a repo.

Commits

PERFORCE
Version everything.

- Git uses commit objects to track units of change
- Each commit object is very simple – it consists of the committer (name + email + timestamp), the author (usually the same as the committer), a tree object representing the state of the repo created by that commit, the commit description, and optionally links to one or more parent commits.
- As of git 1.7.9, commits can also contain the PGP signature of the committer.

Commits

PERFORCE
Version everything.

- Git uses commit entities to track units of change
- Each commit object is very simple – it consists of the committer (name + email + timestamp), the author (usually the same as the committer), a tree object representing the state of the repo created by that commit, the commit description, and optionally links to one or more parent commits.

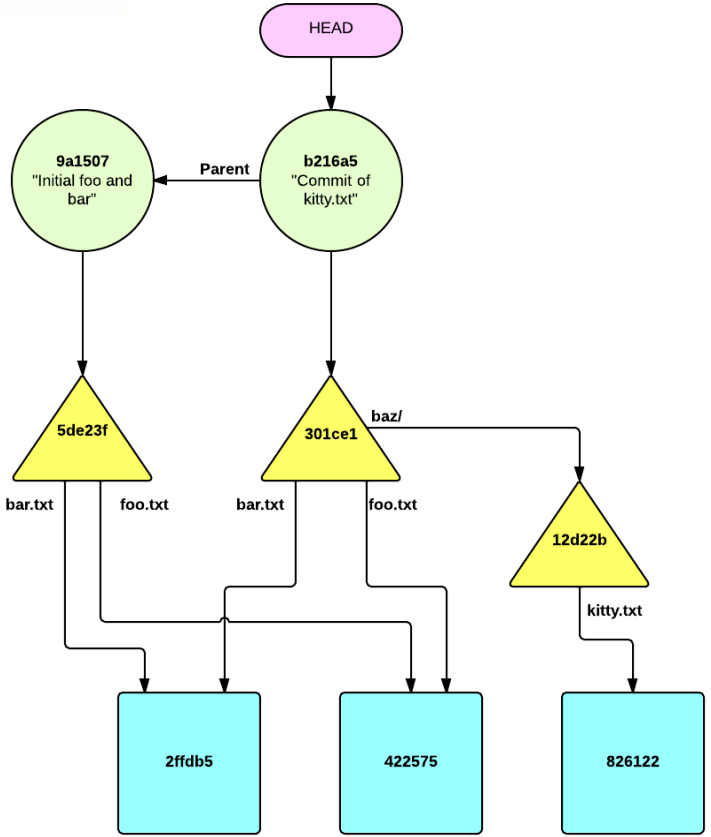
```
➜  simple-2 git:(master) git log -n 1
commit b216a59da1dddae35cbb592552e33cc25365ed9b
Author: ysgard <ysgard@gmail.com>
Date:   Tue Aug 19 09:06:14 2014 -0700

    Commit of kitty.txt
➜  simple-2 git:(master)
```

```
➜  simple-2 git:(master) git cat-file -p b216a5
tree 301ce1a5d189b673b59683af13128cfd675f975b
parent 9a1507465c87653c2f63376786628814f63941ae
author ysgard <ysgard@gmail.com> 1408464374 -0700
committer ysgard <ysgard@gmail.com> 1408479684 -0700

Commit of kitty.txt
➜  simple-2 git:(master)
```

Commits

PERFORCE
Version everything.

- We can use the tree example from earlier to show how commits relate to trees and blobs.
- Each commit points to one tree – the root tree representing the state of the repo after it has been updated with that commit's changes.
- Each commit points to its parent(s), except the first commit, which has none.

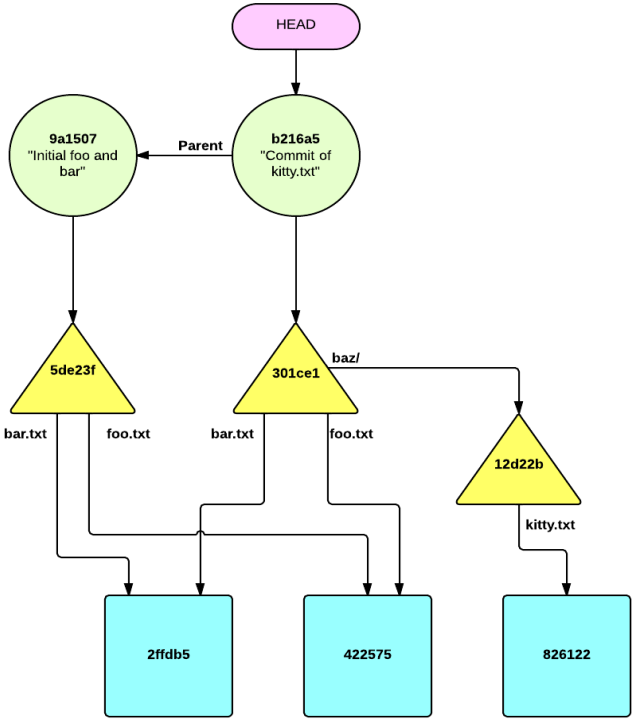


Commits

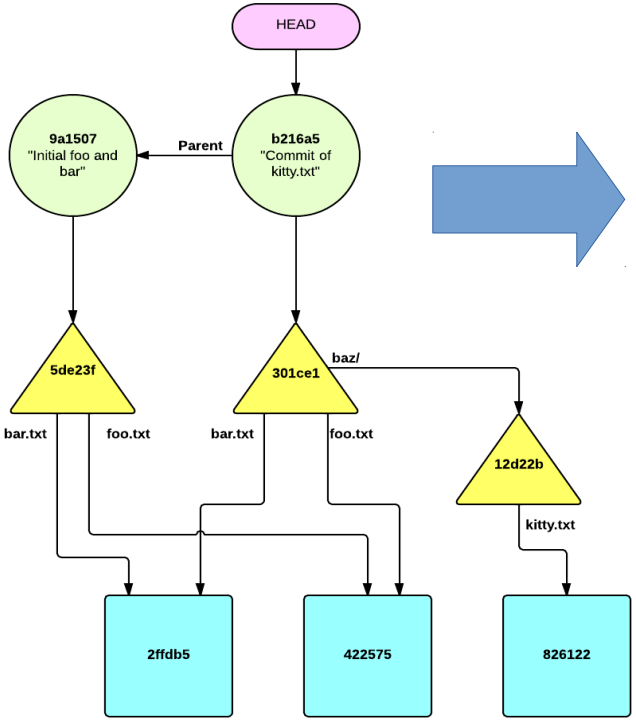
PERFORCE
Version everything.

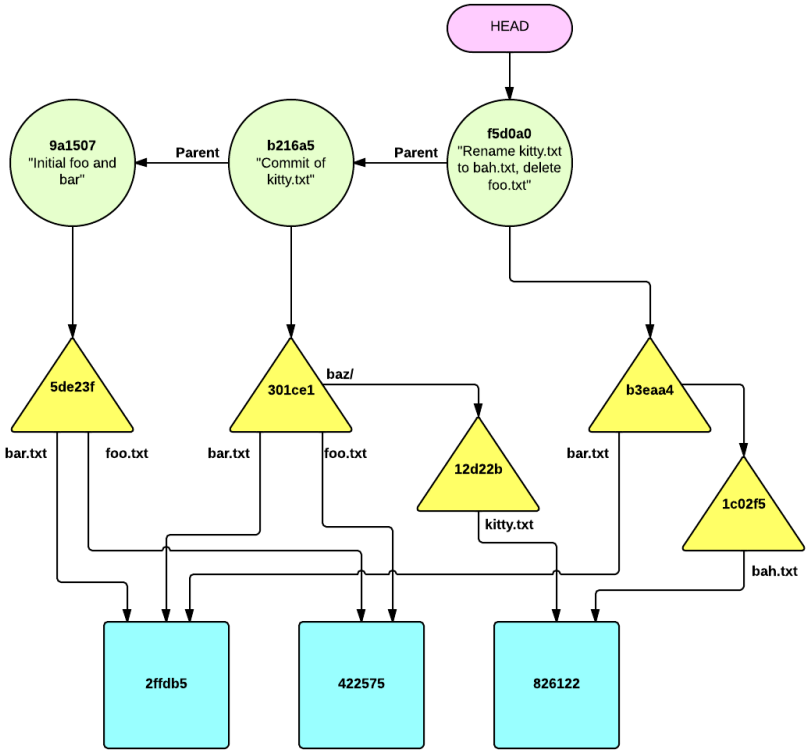
- With every change committed to the repot, new blobs and trees get created to represent new files and arrangements. If a file or directory hasn't changed, then the old trees and objects are used.
- Another example: what happens when we delete foo.txt and rename kitty.txt to bah.txt?

Commits


HEAD
9a1507
"Initial foo and bar"
Parent
b216a5
"Commit of kitty.txt"
5de23f
301ce1
baz/
bar.txt
foo.txt
bar.txt
foo.txt
12d22b
kitty.txt
2ffdb5
422575
826122

Commits

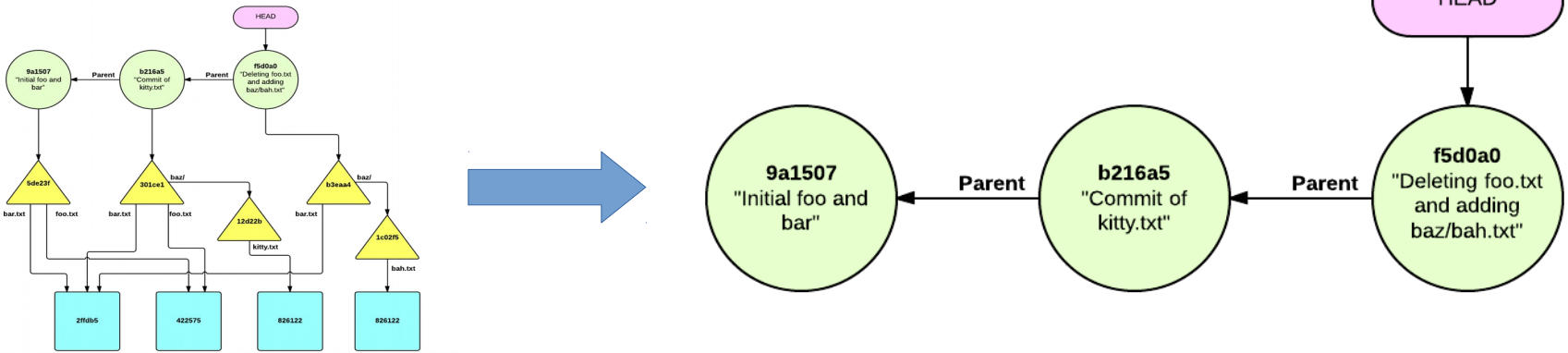

HEAD
9a1507
"Initial foo and bar"
Parent
b216a5
"Commit of kitty.txt"
5de23f
301ce1
baz/
bar.txt
foo.txt
bar.txt
foo.txt
12d22b
kitty.txt
2ffdb5
422575
826122


HEAD
9a1507
"Initial foo and bar"
Parent
b216a5
"Commit of kitty.txt"
Parent
f5d0a0
"Rename kitty.txt to bah.txt, delete foo.txt"
5de23f
301ce1
baz/
b3eaa4
bar.txt
foo.txt
bar.txt
foo.txt
12d22b
kitty.txt
bar.txt
1c02f5
bah.txt
2ffdb5
422575
826122

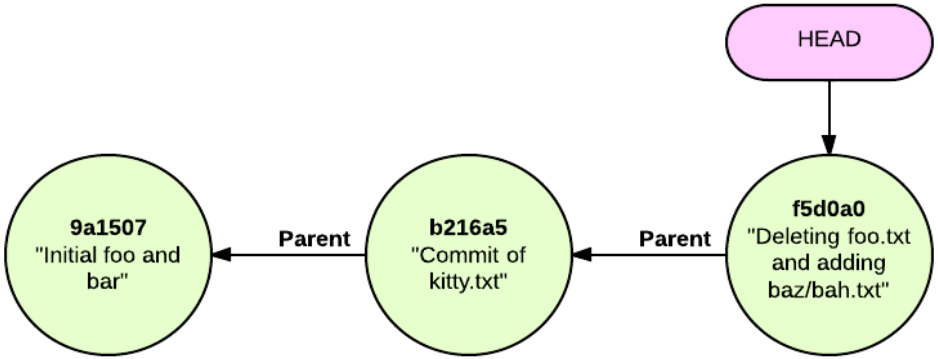
Commits

PERFORCE
Version everything.

- So, trees organize blobs and trees into snapshots of the repo
- Commits point to these snapshots and record information about that commit, as well as pointing to their parents.
- This means that we can represent the internal organization of git purely in terms of commits (and refs).



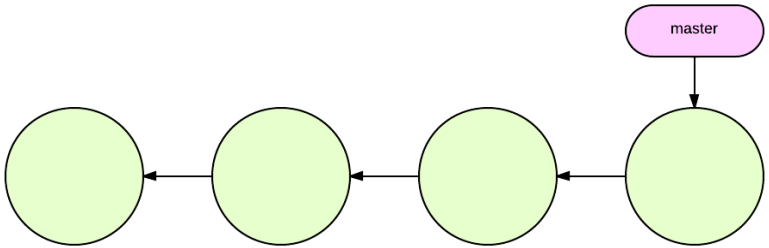
Commits



- Each commit doesn't know about any children – only its parents, of which it can have none (initial commit), one (a normal commit) or more (a merge).

Commits

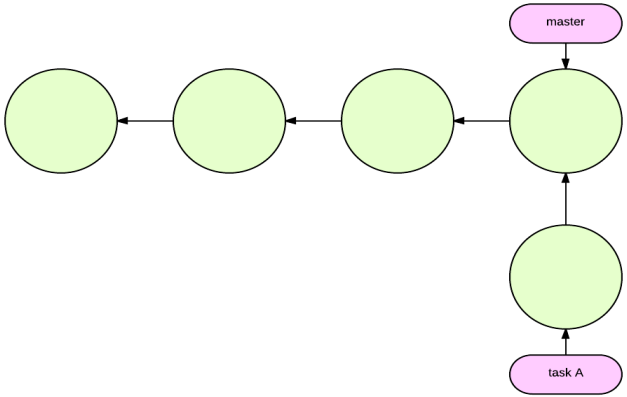
PERFORCE
Version everything.



- Each commit doesn't know about any children – only its parents, of which it can have none (initial commit), one (a normal commit) or more (a merge).

- This kind of structure is called a “directed acyclical graph”, and points to the nature of Git's history.

Commits



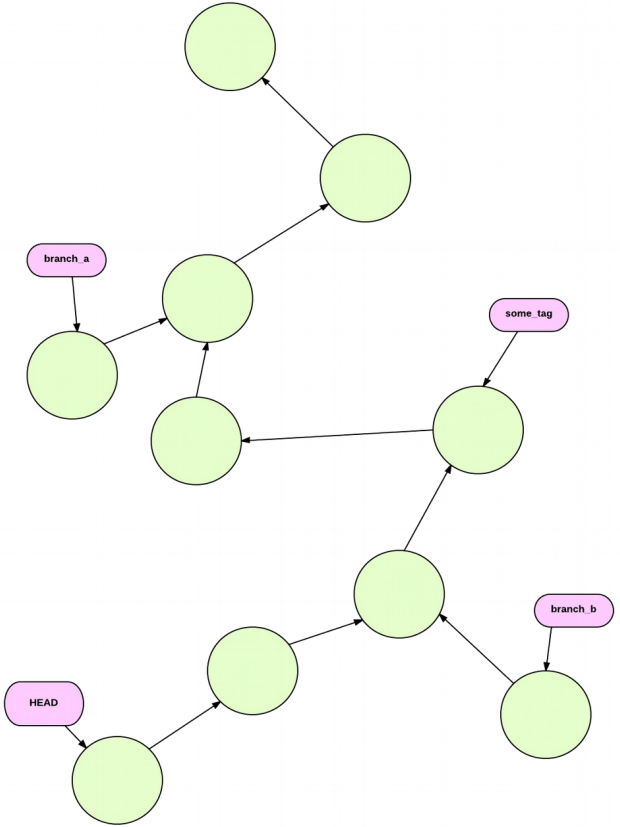
- Each commit doesn't know about any children – only its parents, of which it can have none (initial commit), one (a normal commit) or more (a merge).

- This kind of structure is called a “directed acyclical graph”, and points to the nature of Git's 'history'.
- Git doesn't care about time. Timestamps are recorded for reporting purposes only. All Git cares about is what commits have what parents, and which trees the commits point to.

Commits

PERFORCE
Version everything.

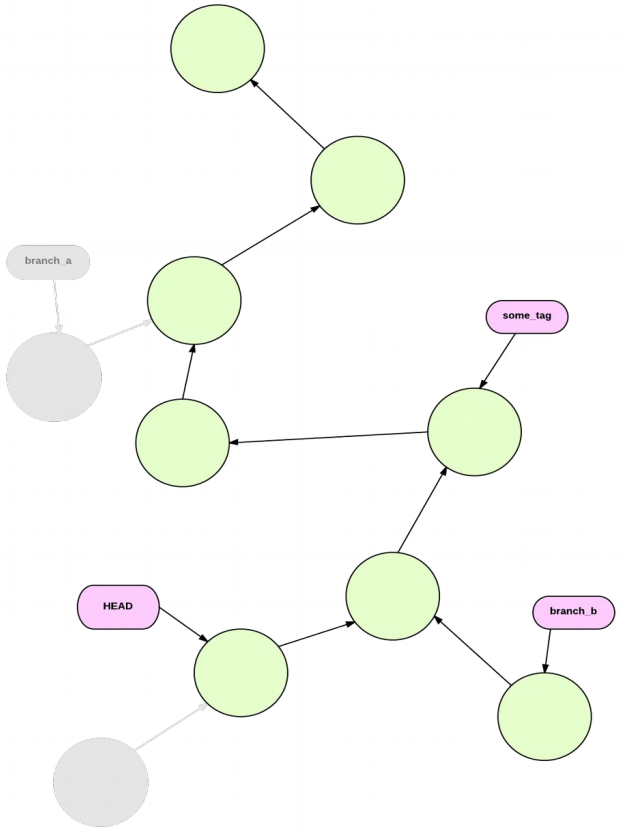
- History in Git is therefore malleable.
- Because the history is just a graph, we can:



Commits

PERFORCE
Version everything.

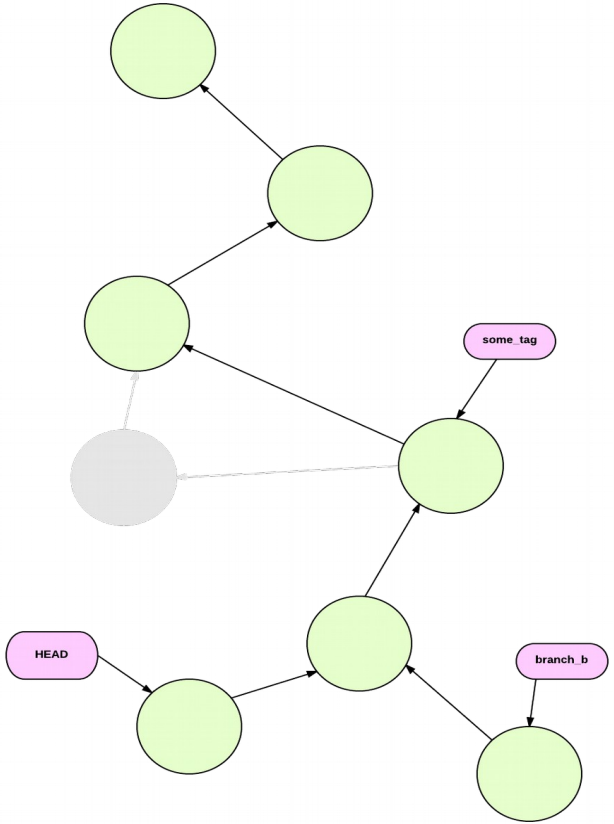
- History in Git is therefore malleable.
- Because the history is just a graph, we can:
  - Prune the graph (git branch -d, git reset –hard)



Commits

PERFORCE
Version everything.

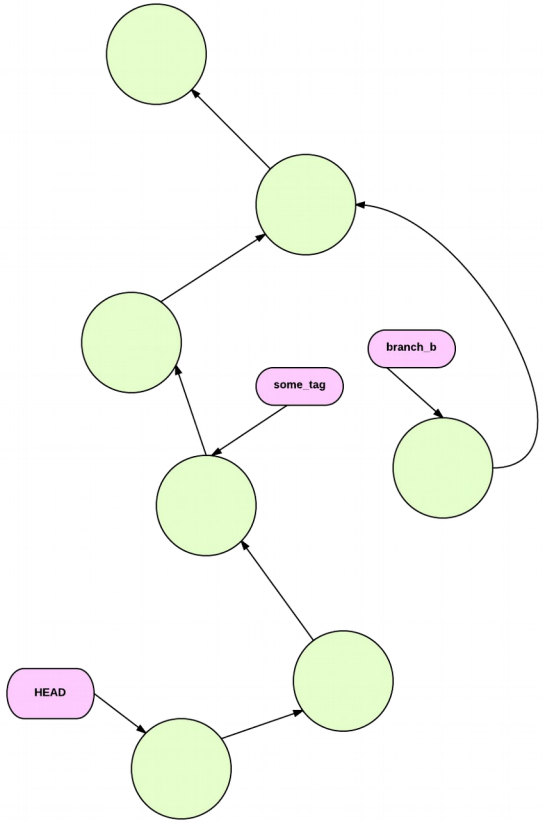
- History in Git is therefore malleable.
- Because the history is just a graph, we can:
  - Prune the graph (git branch -d, git reset –hard)
  - Reorganize the internal nodes (git rebase, git commit –amend)



Commits

PERFORCE
Version everything.

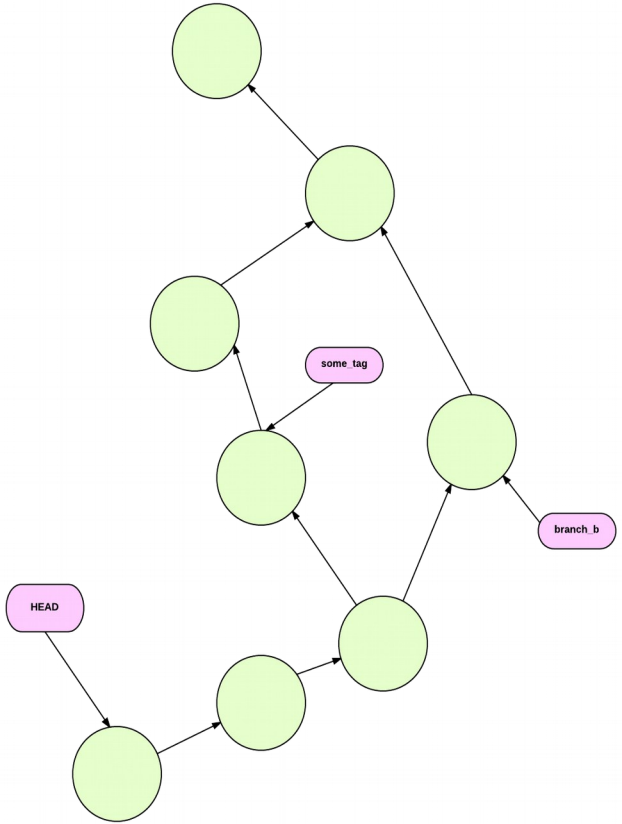
- History in Git is therefore malleable.
- Because the history is just a graph, we can:
  - Prune the graph (git branch -d, git reset –hard)
  - Reorganize the internal nodes (git rebase, git commit –amend)
  - Give commits new parents (git rebase –onto)



Commits

PERFORCE
Version everything.

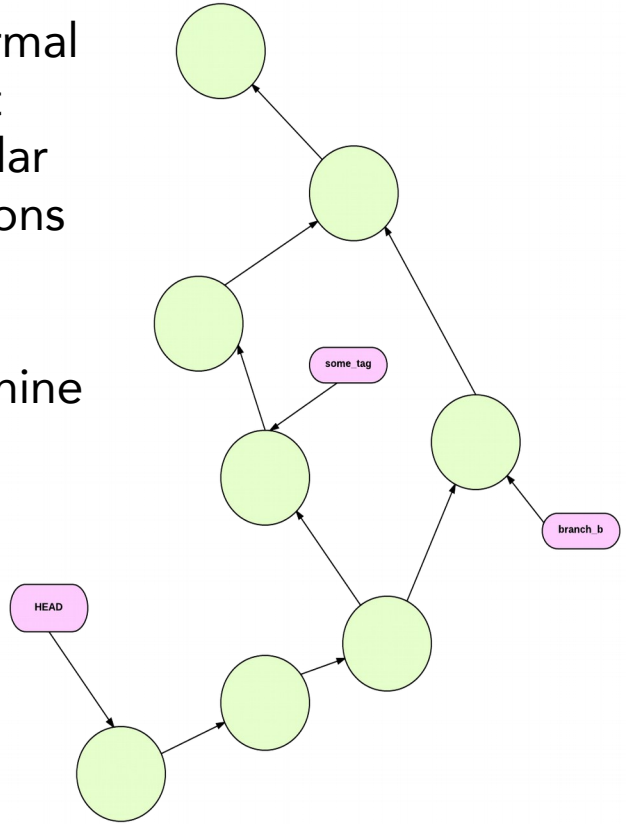
- History in Git is therefore malleable.
- Because the history is just a graph, we can:
  - Prune the graph (git branch -d, git reset –hard)
  - Reorganize the internal nodes (git rebase, git commit –amend)
  - Give commits new parents (git rebase –onto)
  - Create new commits (git commit, git merge)



Commits

PERFORCE
Version everything.

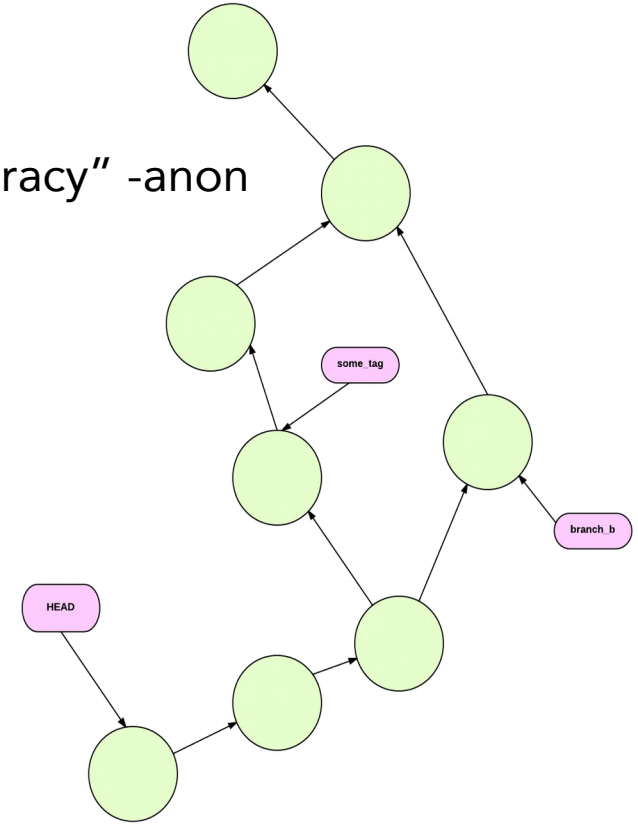
- Manipulating the graph outside of git's normal workflow (which we'll examine in a bit) isn't generally recommended except for particular circumstances, as these internal manipulations can cause troubles for collaborators.
- But Git is flexible and allows you to shoot yourself in the foot, repeatedly, with a machine gun if that's what you want.

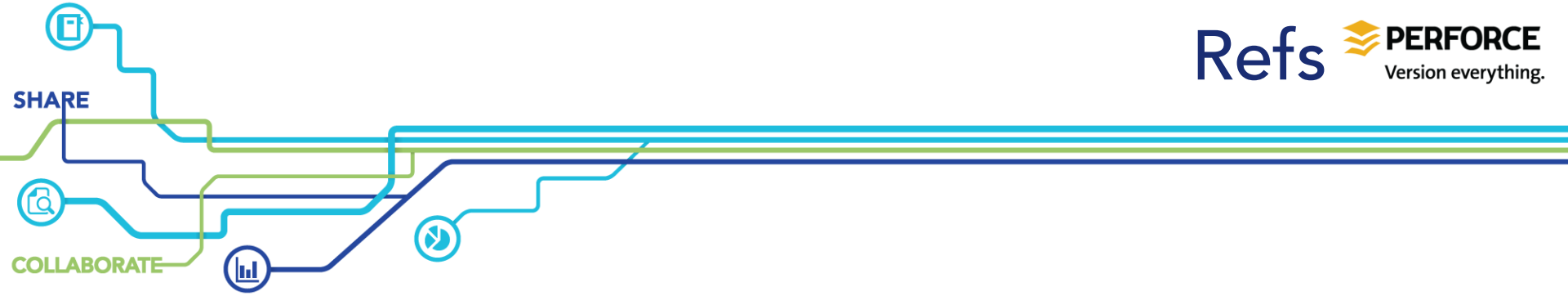


Commits

PERFORCE
Version everything.

"History is the vast and tangled web of conspiracy" -anon





Refs

PERFORCE
Version everything.

- Refs, in comparison to everything else, are simple.
- Refs point to a commit.
- Some refs point to other refs. These are called *symbolic refs*.
- Refs are stored in .git/refs

```
➜ simple git:(master) tree .git/refs
.git/refs
├── heads
│   └── master
└── tags
```

Refs

PERFORCE
Version everything.

- All branches are refs. 'master' is a ref.
- You can use git rev-parse to identify which commit a ref is pointing to:

```
➜  simple git:(master) git rev-parse master
ed0e6e213cdf11330c194530f80d19545dab4842
```

- Git tags are also refs.
- The difference between a branch (aka head) ref and a tag ref is that the head ref(which normally points to the last commit on a branch) is advanced with each new commit, whereas a tag ref always points to the same commit.

Refs

PERFORCE
Version everything.

- Git provides some default refs.
- HEAD*, for example, is a symbolic ref that usually points to the head ref of the current branch you are on.
- ORIG_HEAD contains the previous commit that HEAD had before it changed (because of a merge, a rebase, a commit, etc..)
- Others include FETCH_HEAD and MERGE_HEAD, which we'll see later.

* When HEAD doesn't point to a branch ref, but instead Contains a SHA-1 hash of a commit, then the HEAD is said to be *detached* from any branch.

Refs

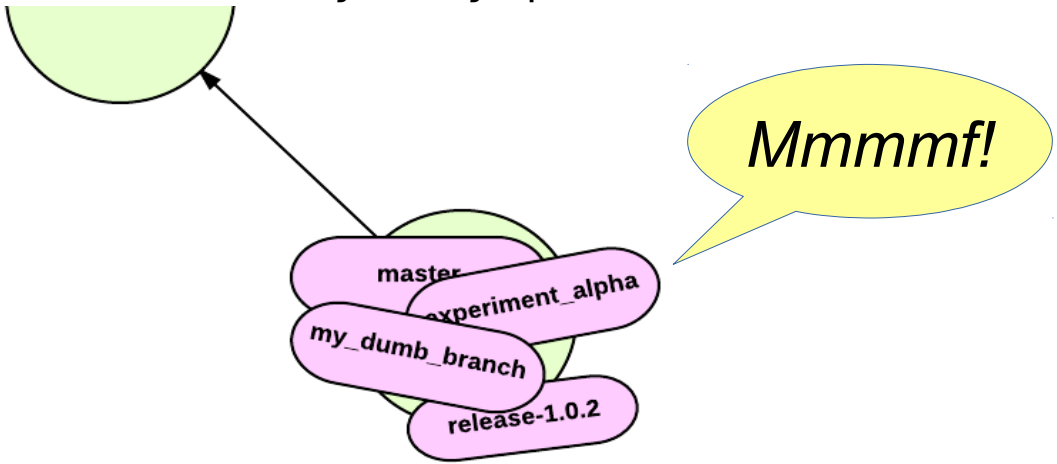
PERFORCE
Version everything.

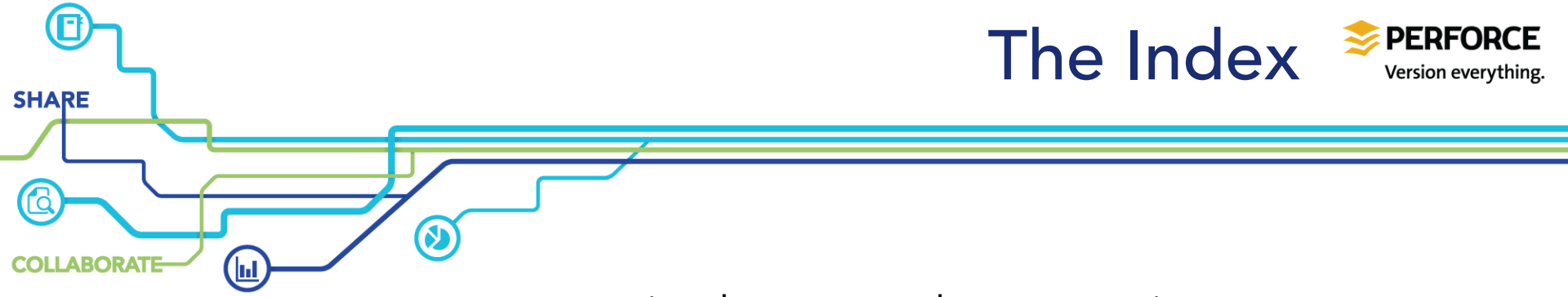
- Most Git commands will accept a ref instead of a sha-1 hash, which makes them invaluable for navigating a Git repo.
- Aside from head refs and tags, there are two other kinds of refs: *stash refs*, which are used to commit objects that are used to store temporary work, and *remote refs*, which contain the heads of the branches of any remote repositories tracked by your repo.

Refs

PERFORCE
Version everything.

- Even though there are different types, all refs are just labels slapped on commits. The commit it points to may change (head ref), not change (tag ref), be temporary (stash ref) or or obtained from another repo (remote ref), but they are still just labels, and they always point to a commit.





The Index

PERFORCE
Version everything.

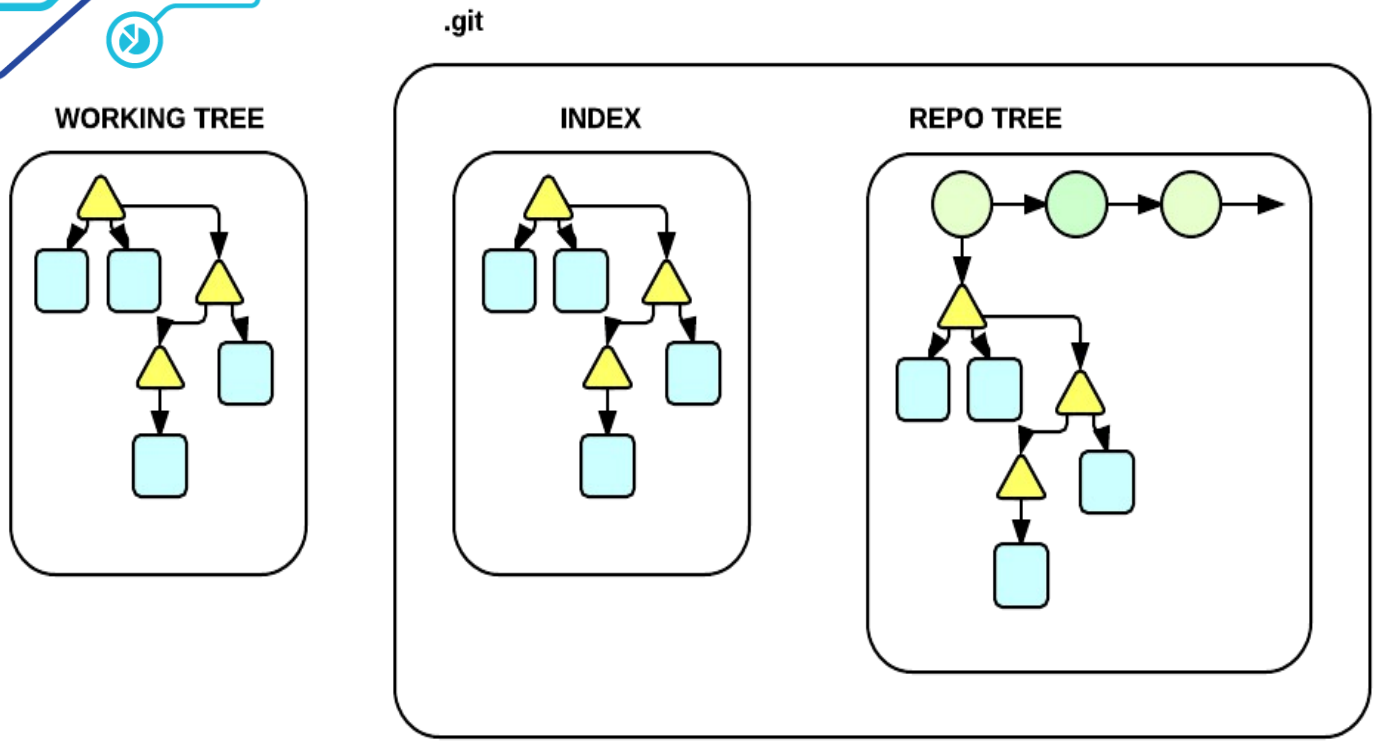
- ...contains the proposed next commit.
- Also known as the *staging area*, because git will refer to files referred to in the index as being *staged*.
- Files added to the index are added to the object store, however they are not associated with any commit.
- So how does Git track the files in the index? It uses a special, volatile tree object stored in *.git/index*

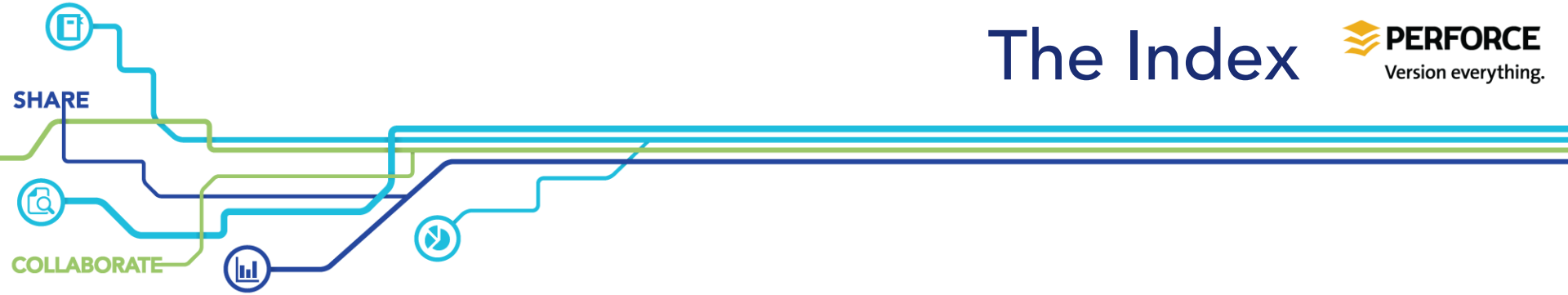
The Index

PERFORCE
Version everything.

SHARE

COLLABORATE





The Index

PERFORCE
Version everything.

- You can see any changes between the index and the working tree using git status
- You can see the actual tree-ish object stored in the index using git ls-files -s

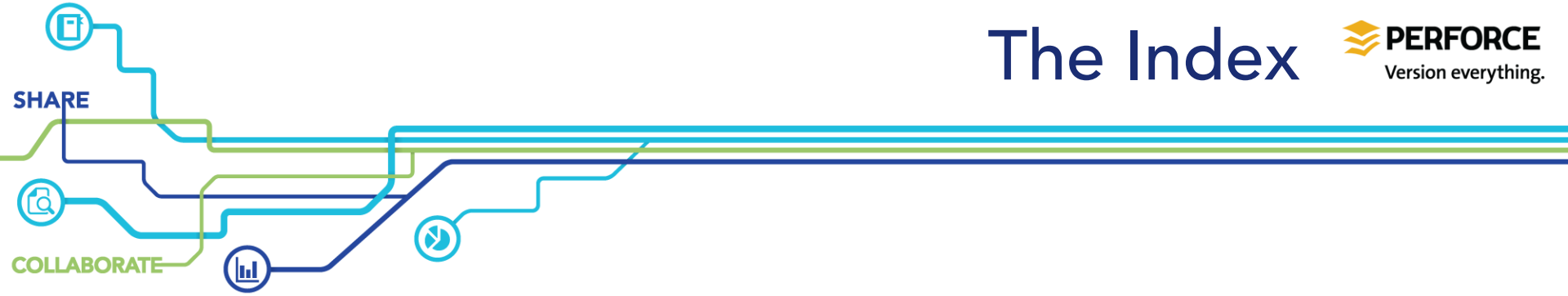
The Index

PERFORCE
Version everything.

- Unlike normal trees, the index stores the whole path instead of creating sub-trees to store nested directories.

```
➜ index-play git:(master) ✗ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   baz/bah.txt
        new file:   foo.txt

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

        bar.txt

➜ index-play git:(master) ✗
```

```
➜ index-play git:(master) ✗ git ls-files --s
100644 78d2d7e902213c81d1f1eb8fea9e766baceb423a 0       baz/bah.txt
100644 1306ea70458d9cdf06c88783c4f0b658126101ea 0       foo.txt
➜ index-play git:(master) ✗
```



The Index

PERFORCE
Version everything.

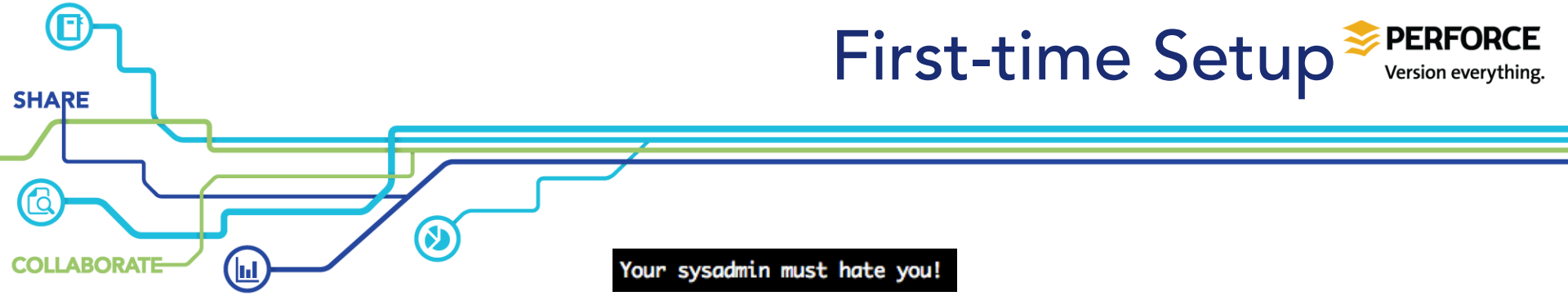
- You can use the index to create arrangements of files and directories, and then write the resulting trees into the object store with plumbing commands such as git update-index, git write-tree, git mk-tree and git commit-tree.
- This allows to to create arbitrary blobs and trees and link them into existing Git repos, allowing you to insert missing history, repair damaged commits or perform any sort of low-level manipulation.

.

PERFORCE
Version everything.

Git Common Tasks

config, init, add, status, commit,
rm, mv, log, branch, checkout,
reset, merge, rebase and tag



First-time Setup

PERFORCE
Version everything.

```
Your sysadmin must hate you!
```

```
You don't exist. Go away!
```

```
Your parents must have hated you!
```

- Before you can start using Git, you need to perform some first-time setup
- At minimum, Git wants to know your name and your email address

git config

PERFORCE
Version everything.

- To set options, you use git config
- Git config can set options on a per-system, per user or per-repo basis.

  `--local` (default) – set option just for this repo
  `--global` – for the user
  `--system` – for the entire system

- So, to set your username and email address for all your git projects:

```
git config --global user.name “Jan Van Uytven”
git config --global user.email “juytven@perforce.com”
```

git config

PERFORCE
Version everything.

- To view your currently set options, use `git config --global --list`

```
➜  git_presentation git:(master) ✗ git config --global --list
user.name=Jan Van Uytven
user.email=juytven@perforce.com
user.signingkey=F8B78E68
core.editor=emacs -nw
core.excludesfile=/Users/wyvern/.gitignore_global
core.pager=
color.ui=true
difftool.kdiff3.path=/usr/local/bin/kdiff3
difftool.kdiff3.trustexitcode=false
difftool.prompt=false
diff.tool=kdiff3
mergetool.kdiff3.path=/usr/local/bin/kdiff3
mergetool.kdiff3.trustexitcode=false
mergetool.keepbackup=false
merge.tool=kdiff3
alias.lg=log --color --graph --pretty=format:'%C(red)%h -%C(yellow)%d  %C(white)%s %C(
green)(%cr) %C(bold blue)<%an>%Creset'
alias.co=checkout
alias.ds=diff --staged
alias.st=status -sb
alias.amend=commit --amend
```

git config

PERFORCE
Version everything.

- Other useful (but optional) variables to set:
  - core.editor
    This is the default editor git will use when requesting commit messages. When setting the editor, remember to include a 'wait' flag, if the editor supports one. Sample configurations:

```
$ git config --global core.editor “<editor>”
```

where <editor> is...

- Sublime Text: subl -w
- TextMate: mate -w
- Gvim, MacVim: gvim -f, mvim -f
- Atom: atom -nw
- Emacs: emacs -nw

git config

PERFORCE
Version everything.

- Other useful (but optional) variables to set:
  - core.pager
    By default, git will call a pager (more, less or cat, depending on your system) to handle the output of commands like git diff, git log, etc... This can be helpful or a real pain, depending on your workflow. To turn off the pager*, set it to a blank string.

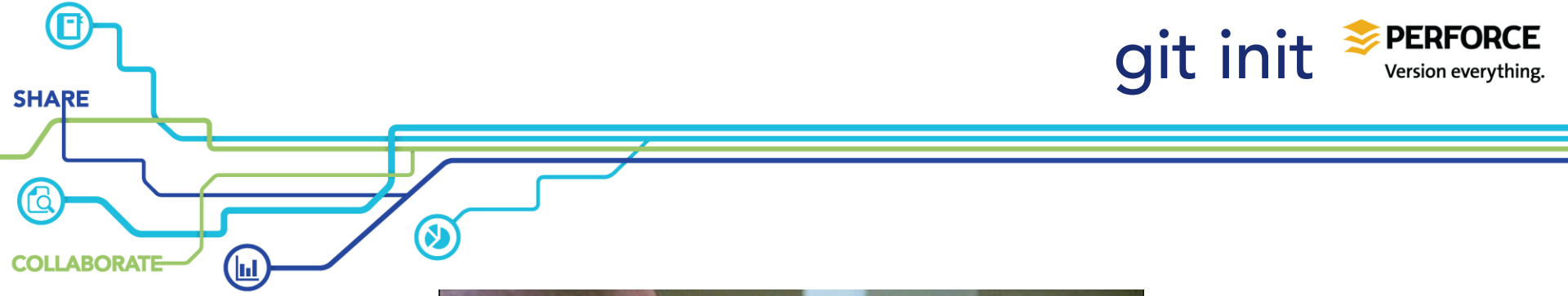
```
$ git config --global core.pager ''
```

* You can also disable it on a per-command basis using –no-pager

git config

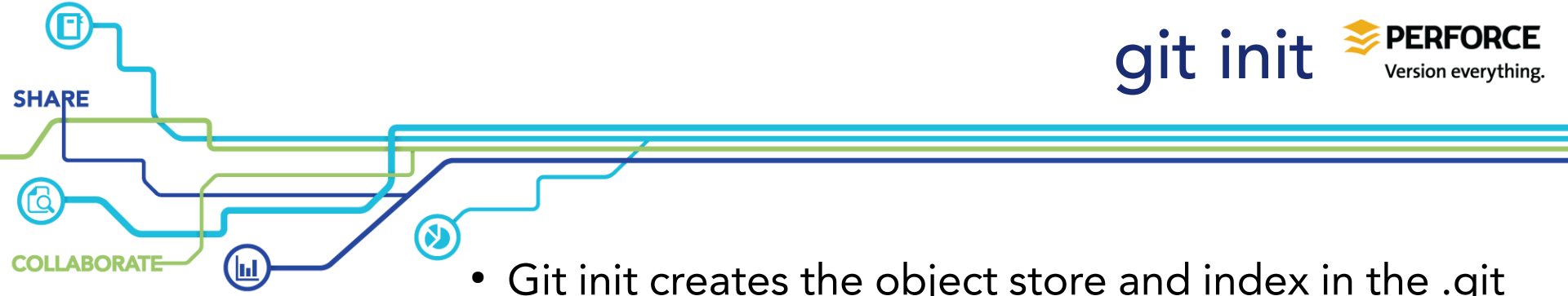
PERFORCE
Version everything.

- Other useful (but optional) variables to set:
  - diff.tool, merge.tool
    Sets the default diff and merge tools used by git difftool and git mergetool, respectively.
  - user.signingkey
    If you want to use PGP/GPG to sign your tags and commits, set this to your key id.



git init

PERFORCE
Version everything.



git init

PERFORCE
Version everything.

- Git init creates the object store and index in the .git subdirectory with the assumption that the repo is going to be used by a user

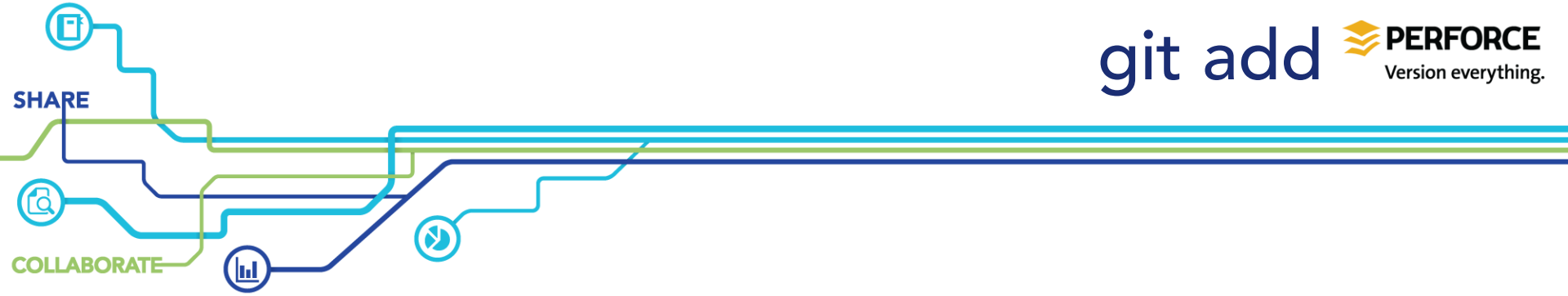
```
➜ git-init  ls -al
total 0
drwxr-xr-x  2 juytven  staff   68 21 Aug 09:11 .
drwxr-xr-x  9 juytven  staff  306 21 Aug 09:11 ..
➜ git-init  git init
Initialized empty Git repository in /Users/juytven/Dropbox/git-examples/git-init/.git/
➜ git-init git:(master) ls -al
total 0
drwxr-xr-x  3 juytven  staff  102 21 Aug 09:11 .
drwxr-xr-x  9 juytven  staff  306 21 Aug 09:11 ..
drwxr-xr-x  9 juytven  staff  306 21 Aug 09:11 .git
➜ git-init git:(master)
```

git init

PERFORCE
Version everything.

- It is possible to create a git repo with no index and no working directory, solely for the purposes of storing changes from other repos.
- To create this kind of repo, use the command
  git init --bare

```
➜  bare-example  git init --bare
Initialized empty Git repository in /Users/juytven/Dropbox/git-examples/bare-example/
➜  bare-example git:(master) ls -al
total 24
drwxr-xr-x   9 juytven  staff  306 21 Aug 13:05 .
drwxr-xr-x  10 juytven  staff  340 21 Aug 13:05 ..
-rw-r--r--   1 juytven  staff   23 21 Aug 13:05 HEAD
-rwxr--r--   1 juytven  staff  111 21 Aug 13:05 config
-rw-r--r--   1 juytven  staff   73 21 Aug 13:05 description
drwxr-xr-x  11 juytven  staff  374 21 Aug 13:05 hooks
drwxr-xr-x   3 juytven  staff  102 21 Aug 13:05 info
drwxr-xr-x   4 juytven  staff  136 21 Aug 13:05 objects
drwxr-xr-x   4 juytven  staff  136 21 Aug 13:05 refs
➜  bare-example git:(master)
```



git add

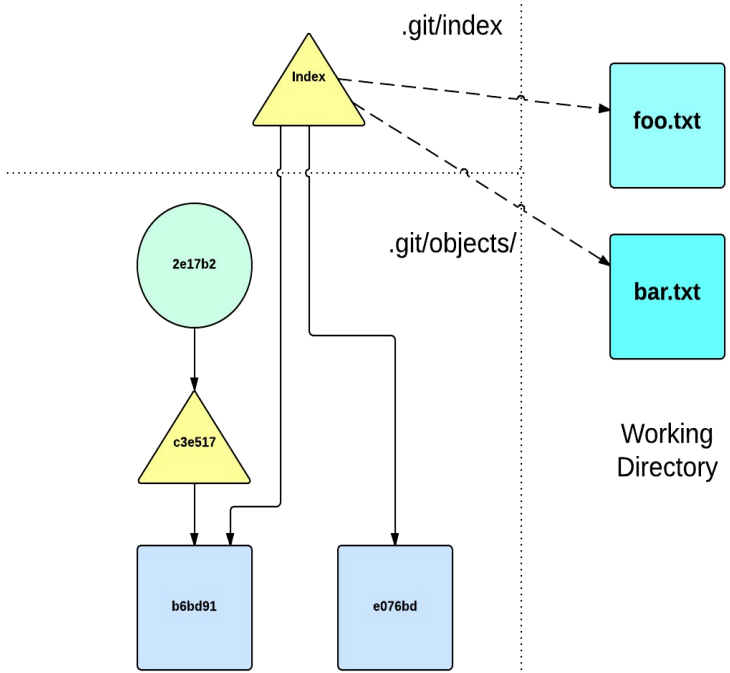
PERFORCE
Version everything.

- Git add is used to add files to the index (which, remember, is the commit we're working towards)
- Adding a file to the index adds it to the object store as well, since the index is a tree and needs to be able to point to a blob containing the file contents.

git add

PERFORCE
Version everything.

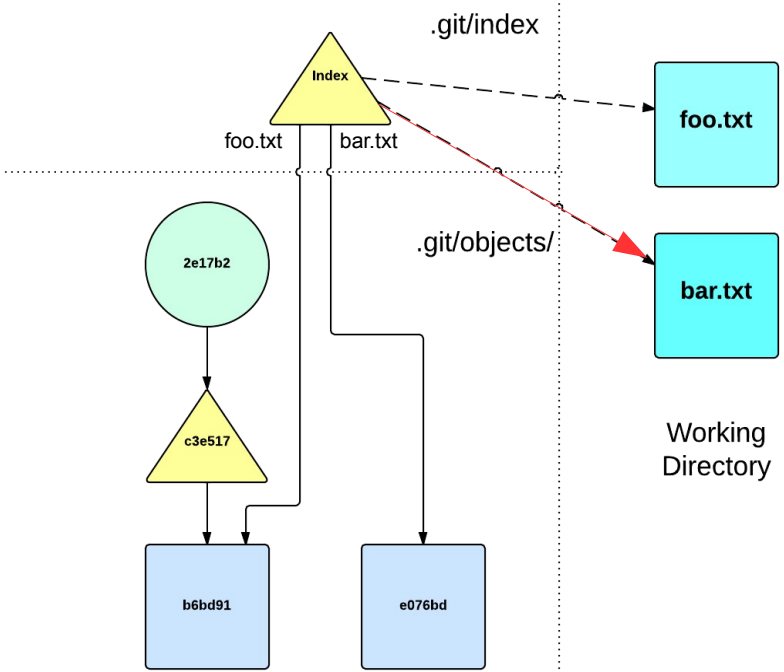
Prior to commit



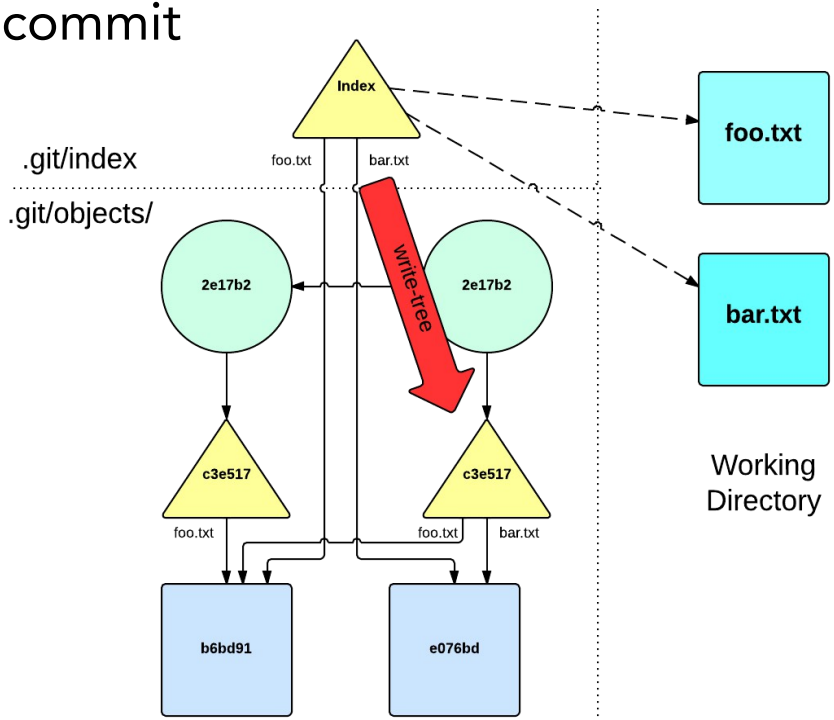
git add

PERFORCE
Version everything.

Prior to commit



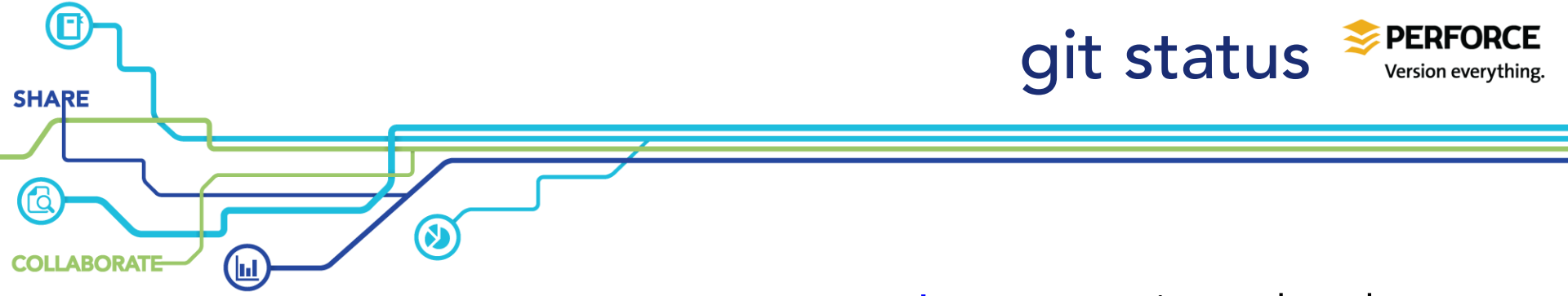
After commit



git add

PERFORCE
Version everything.

- Can use git add to add whole directories. Note that if files are already stored in git, then those files will be ignored by git add.
- `git add .`
  This will add any new and modified files in the current directory, and any subdirectories, to the index.
- `git add -A`
  Adds everything, and also changes the state of the index to exactly match what's in the working directory – this includes deleting files from the index as needed.

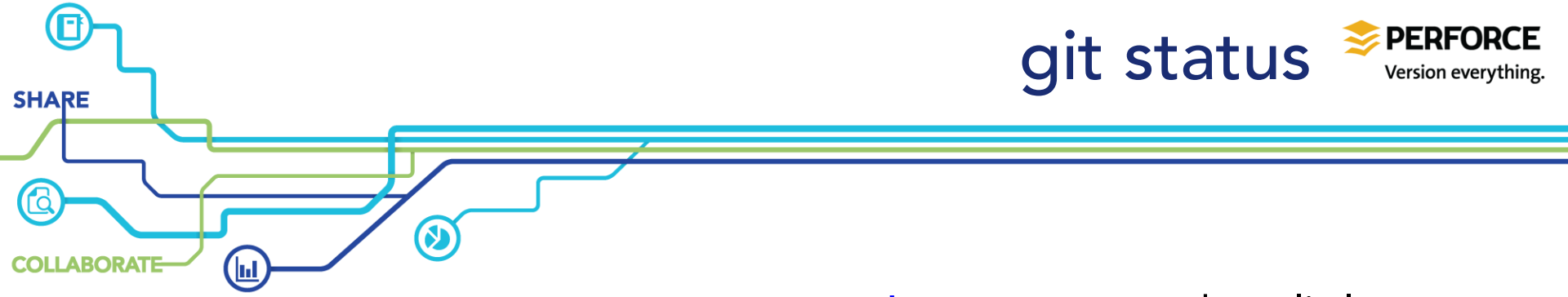


git status

PERFORCE
Version everything.

```
➜ status-example git:(master) ✗ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   bender.txt
        renamed:    leela.txt -> captain_leela.txt
        deleted:    fry.txt

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

        zoidberg.txt

➜ status-example git:(master) ✗
```

- `git status` is used to show the current state of the index
- Shows which files are being added, removed or modified in the index
- Also shows untracked (that is, unmanaged) files
- Provides hints as to how to adjust the index



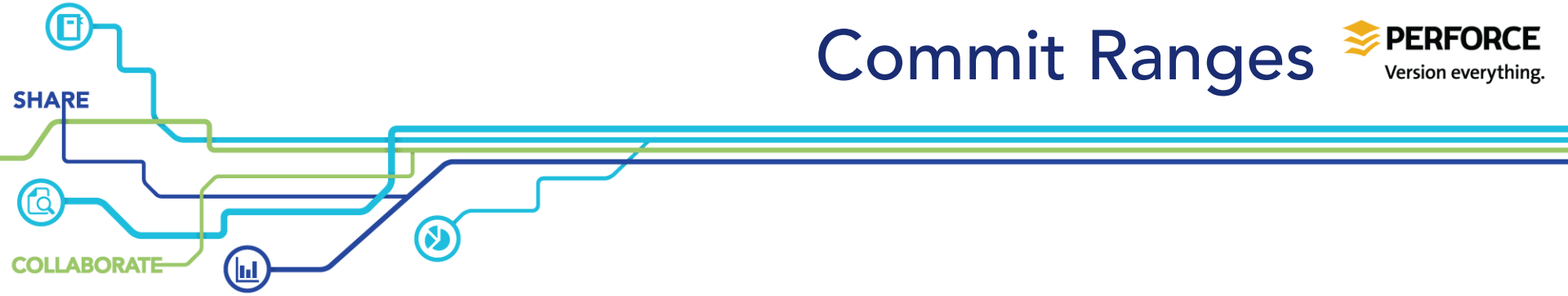
git status

PERFORCE
Version everything.

```
➜  status-example git:(master) ✗ git status -sb
## master
A  bender.txt
R  leela.txt -> captain_leela.txt
D  fry.txt
?? zoidberg.txt
➜  status-example git:(master) ✗
```

- `git status` can be a little verbose, so adding the `-s` (short format) and `-b` (keep branch info in short format) will result in a more concise list: `git status -sb`

- In the rest of this presentation, you may see `git st` used in examples – this is an alias for git status -sb, made using `git config –global alias.st “status -sb”`

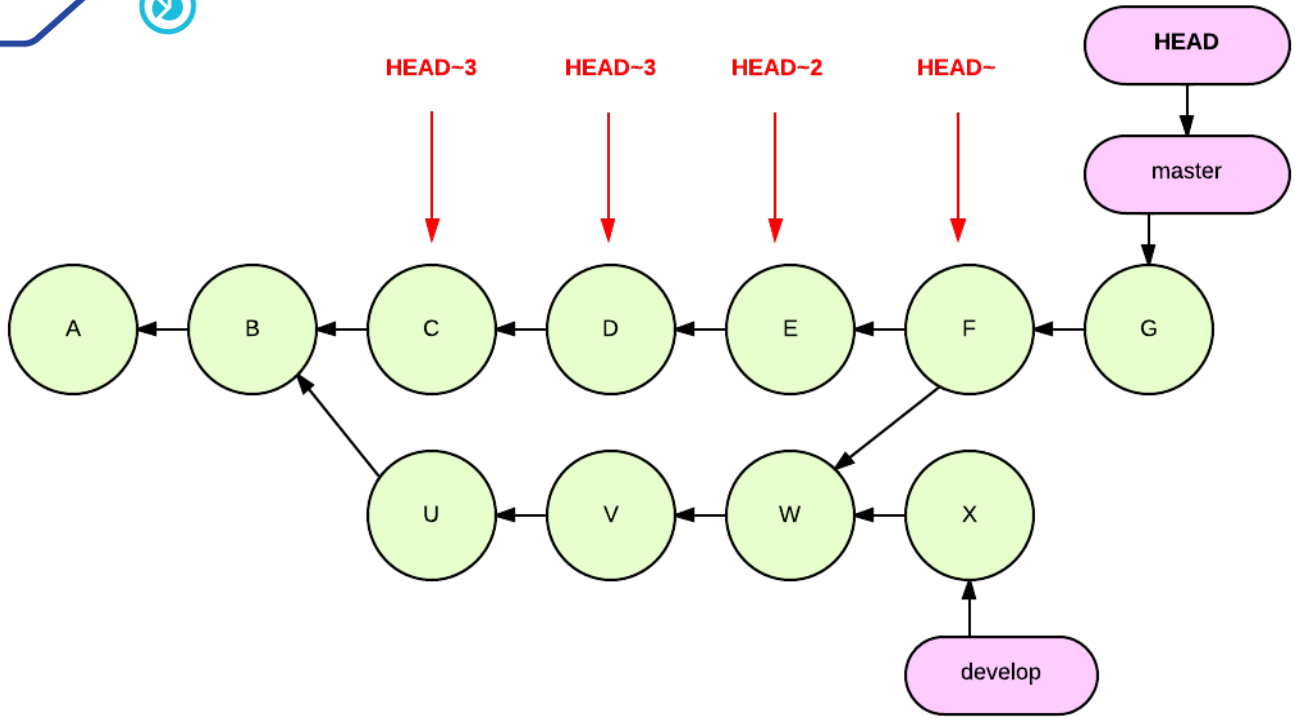


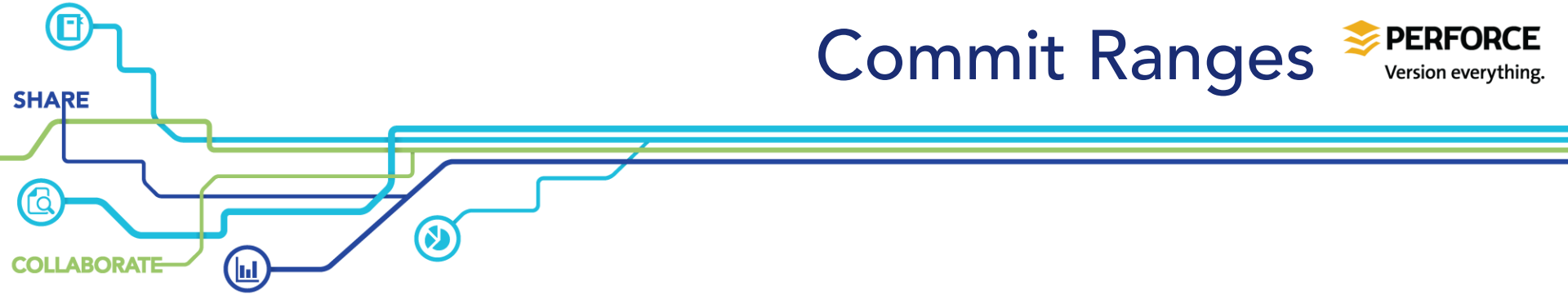
Commit Ranges

PERFORCE
Version everything.

- Git has a flexible method of referring to commits relative to another commit, as well as defining sets of commits.
- We won't go into details (it's *very* flexible) but we will outline some of the more common formats.

Relative Refs





Commit Ranges

PERFORCE
Version everything.

- A commit range takes the form X..Y
- This is read as “All the commits reachable from Y, but not including X and all commits reachable by X.”, where X and Y are either commits or refs.
- For example,
  `topic..master`
  is the set of all commits reachable by the head of master but not including any commits reachable by the head of topic.

Commit Ranges

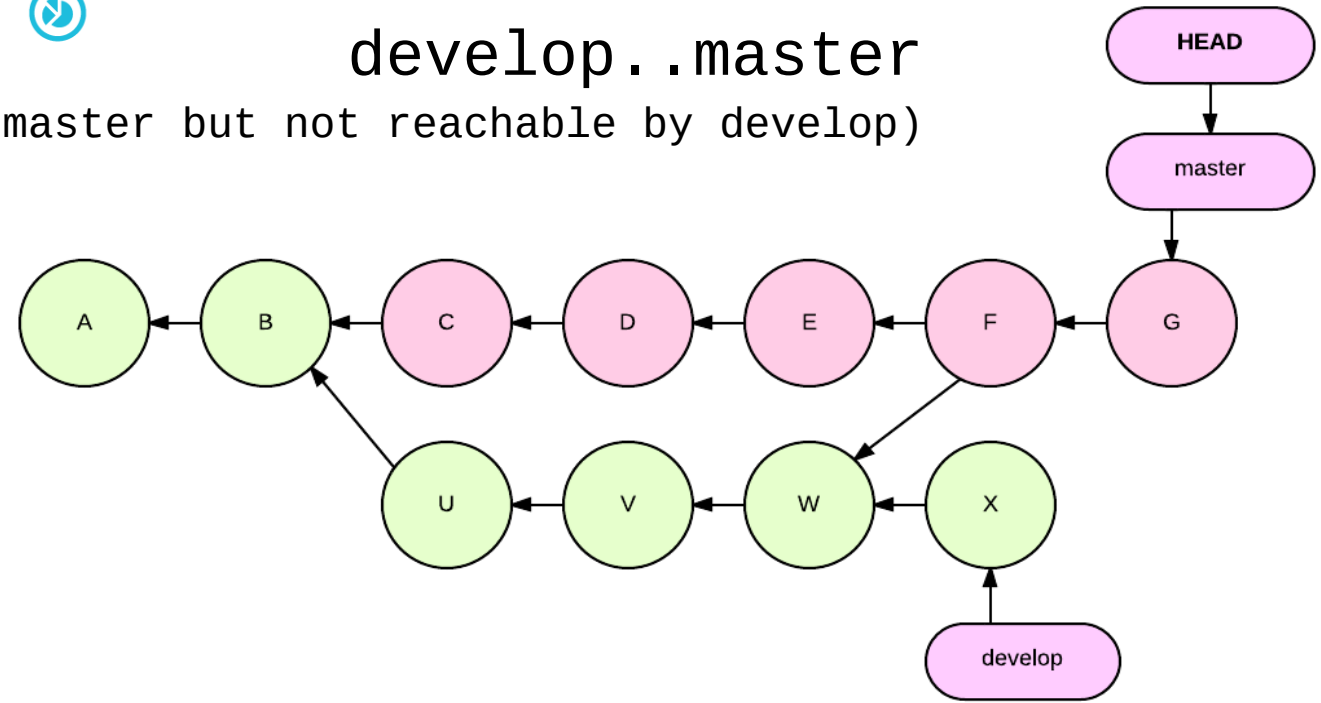
PERFORCE
Version everything.

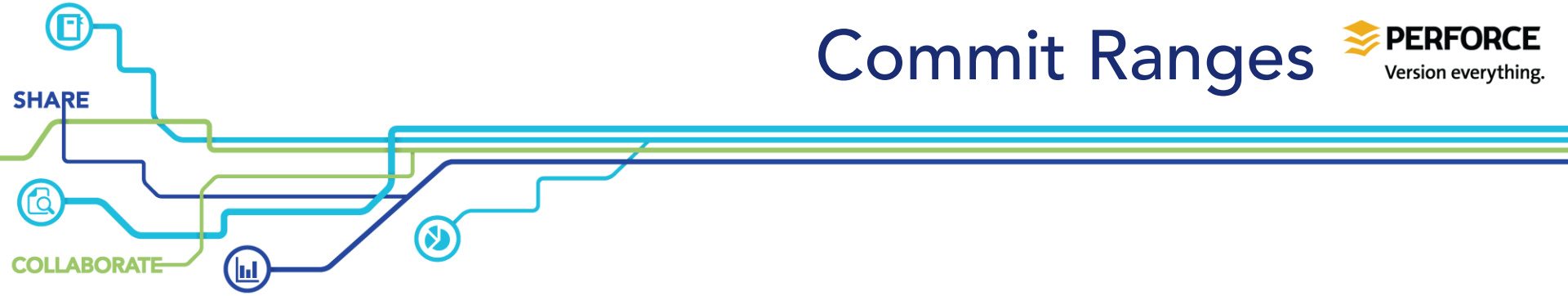
SHARE

COLLABORATE

develop..master

(in master but not reachable by develop)





Commit Ranges

PERFORCE
Version everything.

- Alternatively, you can specify a commit range using sets, and use logical operators like ^ (NOT), OR and AND.
- For example,
  `^dev ^topic ^bugfix master`
  is the set of all commits reachable by master but not including any reachable by dev, topic or bugfix.

git log

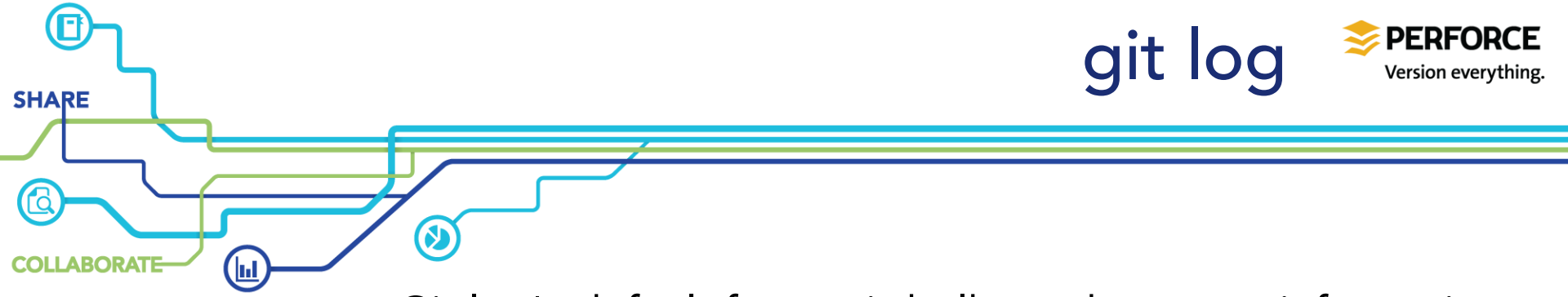
PERFORCE
Version everything.

```
➜  cow-sheep git:(master) git log HEAD~2..HEAD
commit 556e52ed9ea348fcc12c36acd8f36a0c0c6a8e9a
Author: Jan Van Uytven <juytven@perforce.com>
Date:   Wed Aug 27 07:09:29 2014 -0700

    master 3

commit 319764aa2e732ffb48e2b041e966a16edd51a060
Author: Jan Van Uytven <juytven@perforce.com>
Date:   Wed Aug 27 07:09:28 2014 -0700

    master 2
➜  cow-sheep git:(master)
```

- Use git log to provide a full list of every commit in the repo, a commit range, or just a single commit.
  Ex:

  `git log HEAD~2..HEAD`

- This provides a log of the last two commits done on HEAD.



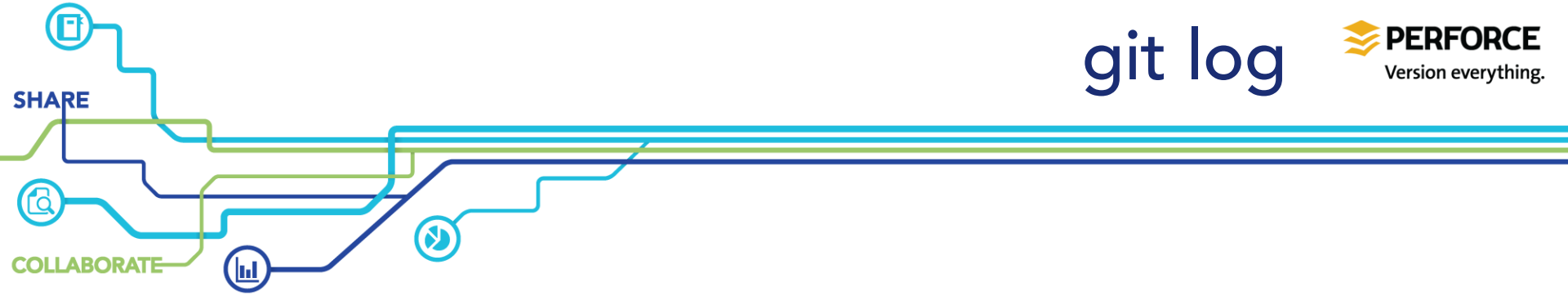
git log

PERFORCE
Version everything.

- Git log's default format is bulky and not very informative
- There is a wealth of options to customize git log, which can turn out some much nicer (and more dense) information:

```
git log –color –graph –pretty=format:'%C(red)%h -%C(yellow)%d %C(white)%s
%C(green)(%cr) %C(bold blue)<%an>%Creset' HEAD~2..HEAD
```

```
➜  cow-sheep git:(master) git lg HEAD~2..HEAD
* 556e52e - (HEAD, master)  master 3 (20 minutes ago) <Jan Van Uytven>
* 319764a -  master 2 (20 minutes ago) <Jan Van Uytven>
➜  cow-sheep git:(master)
```



git log

PERFORCE
Version everything.

- `--graph` provides an ASCII representation of the git repo graph
- `--oneline` gives you one line per commit
- `--pretty='<format_string>'` allows you to customize the format using a formatting string
- `-n<count>` gives you n commits from the output.
- `--since '<time_ago>'` allows for reporting commits made over a specific time period. This is pretty flexible, and can include ranges like '2 months ago', 'today', 'four weeks ago', etc..
- Many more. `git help log` is worth a read.

git standup

PERFORCE
Version everything.

```
➜ gitlab git:(master) git log --since '1 day ago' --oneline --author="Dmitriy"
e38081e More entries to CHANGELOG
d4d723c Merge pull request #7616 from yglukhov/fix-parallel-diff-on-new-mr
c4a8663 Tell travis what directory cache explicitly
a5eb691 Enable bundler caching for travis
f12fdf1 Merge branch 'improve-search-page' into 'master'
67d96b0 Merge pull request #7611 from yglukhov/sticky-diff-headers
f089fd5 Merge pull request #7614 from yglukhov/toggle-line-wrap
c5c906f Fix tests
7f4b993 Forgot to save file :)
043f275 Skip description if not exist
8b00d01 Search by issue/mr title and description
7e59a8f Improve comment search results
13f6dc1 Save search options when switch between filter
c3ad51a Improve search tests
ede08db Implement search page with filtering of results and pagination
9e5bc43 Pass scope and page to Gitlab::SearchResults#objects instead of initialize
5d9a5c0 Add search method to Note class
9a4ef7e Search results libraries added
b3b50bb Merge branch 'master' of dev.gitlab.org:gitlab/gitlabhq
f9a1163 Merge branch 'test/fix_home_dir'
eabbb7b Merge branch 'session_timeout' into 'master'
➜ gitlab git:(master)
```

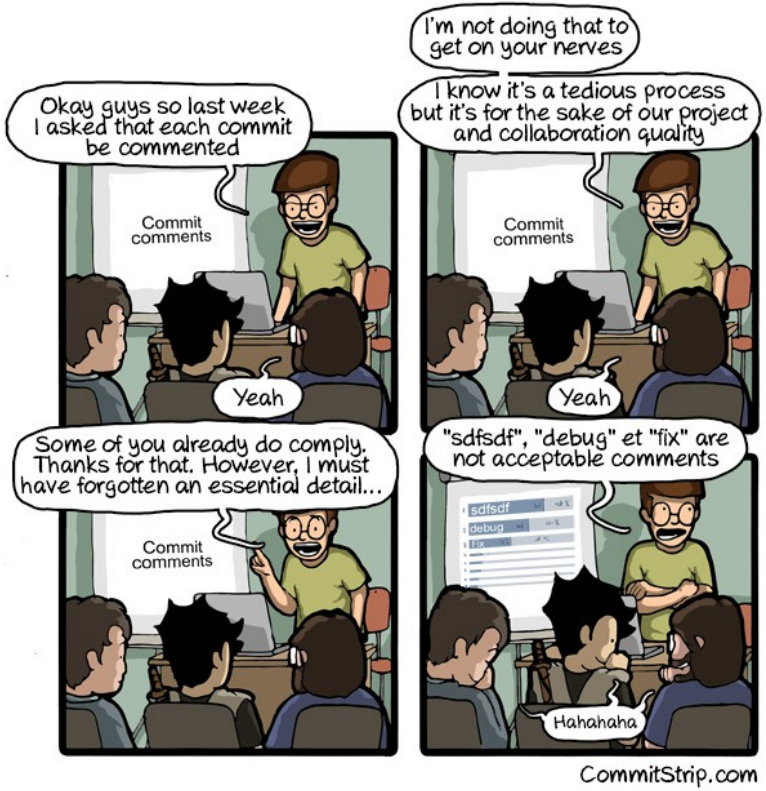
- Last example – using git log to generate a daily standup report
- `git log --since '1 day ago' --oneline --author=Dmitriy`
- Alias it to 'git standup':

```
git config –global alias.standup “log –since '1 day ago' oneline –author=Dmitriy”
```

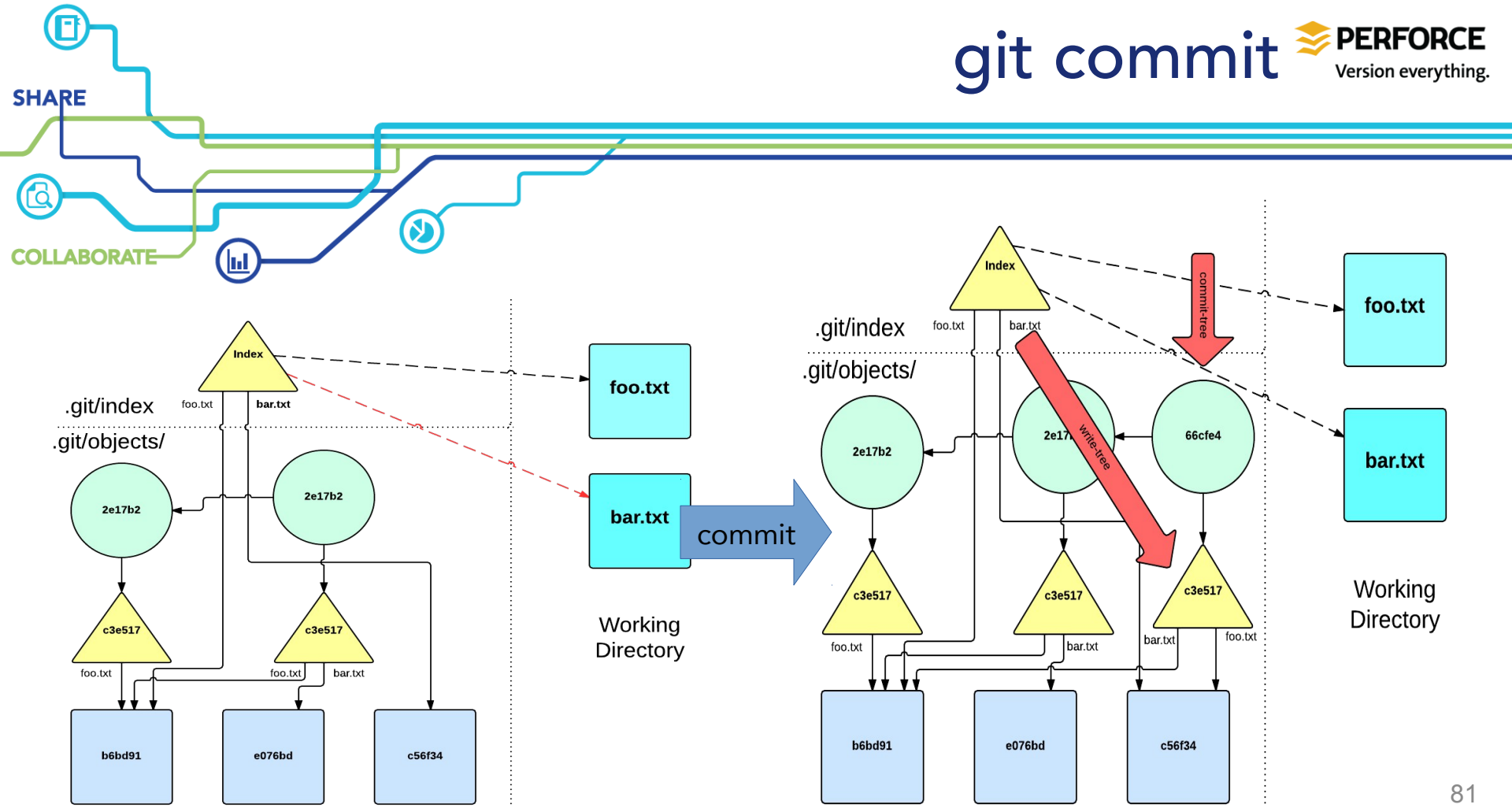
git commit

- Writes out the index to a tree or trees in the object store, then creates a commit object to link to the new root tree and to the prior commit.

- Also stores the name, email and a few other pieces of data, such as the commit description.



git commit

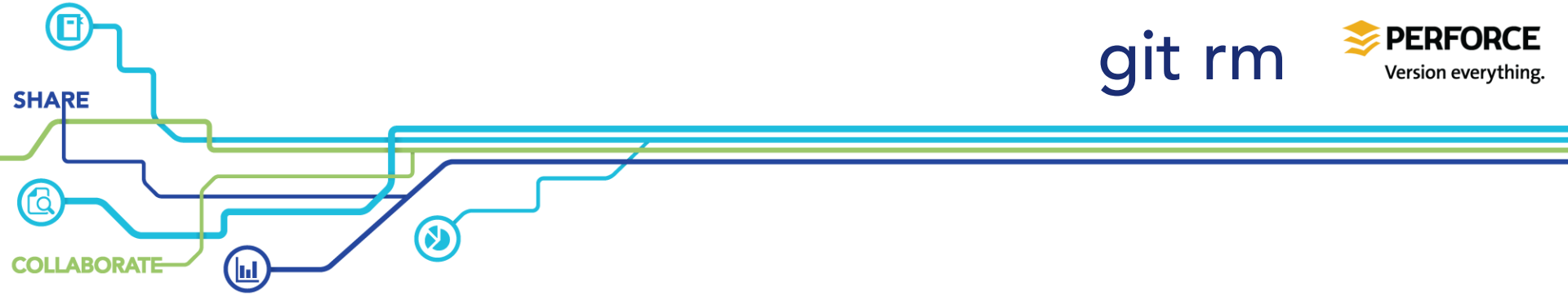

SHARE
COLLABORATE
Index
.git/index
foo.txt
bar.txt
.git/objects/
2e17b2
2e17b2
c3e517
c3e517
foo.txt
foo.txt
bar.txt
b6bd91
e076bd
c56f34
foo.txt
bar.txt
Working Directory
commit
Index
.git/index
foo.txt
bar.txt
.git/objects/
commit-tree
write-tree
2e17b2
66cfe4
c3e517
c3e517
c3e517
foo.txt
bar.txt
bar.txt
foo.txt
b6bd91
e076bd
c56f34
foo.txt
bar.txt
Working Directory

git commit

PERFORCE
Version everything.

```
➜ commit-example git:(master) ✗ git commit -S -m "Committing bar.txt"

You need a passphrase to unlock the secret key for
user: "Jan Van uytven <ysgard@gmail.com>"
4096-bit RSA key, ID F8B78E68, created 2014-06-05

[master 20873d7] Committing bar.txt
 1 file changed, 5 insertions(+), 3 deletions(-)
➜ commit-example git:(master) git show --pretty=fuller 2087
commit 20873d70f6a6a7231b47bc6923c05b20bcd35e37
Author:     Jan Van Uytven <juytven@perforce.com>
AuthorDate: Mon Aug 25 10:22:43 2014 -0700
Commit:     Jan Van Uytven <juytven@perforce.com>
CommitDate: Mon Aug 25 10:22:43 2014 -0700

    Committing bar.txt

diff --git a/bar.txt b/bar.txt
index 183c812..c9febe1 100644
```

- You can get details on a given commit with `git show <commit>`
- You can avoid having to pop up an editor for short commit messages by using `-m` to provide a message.
- If you are using a GPG key, use `-S` to sign the commit with your key

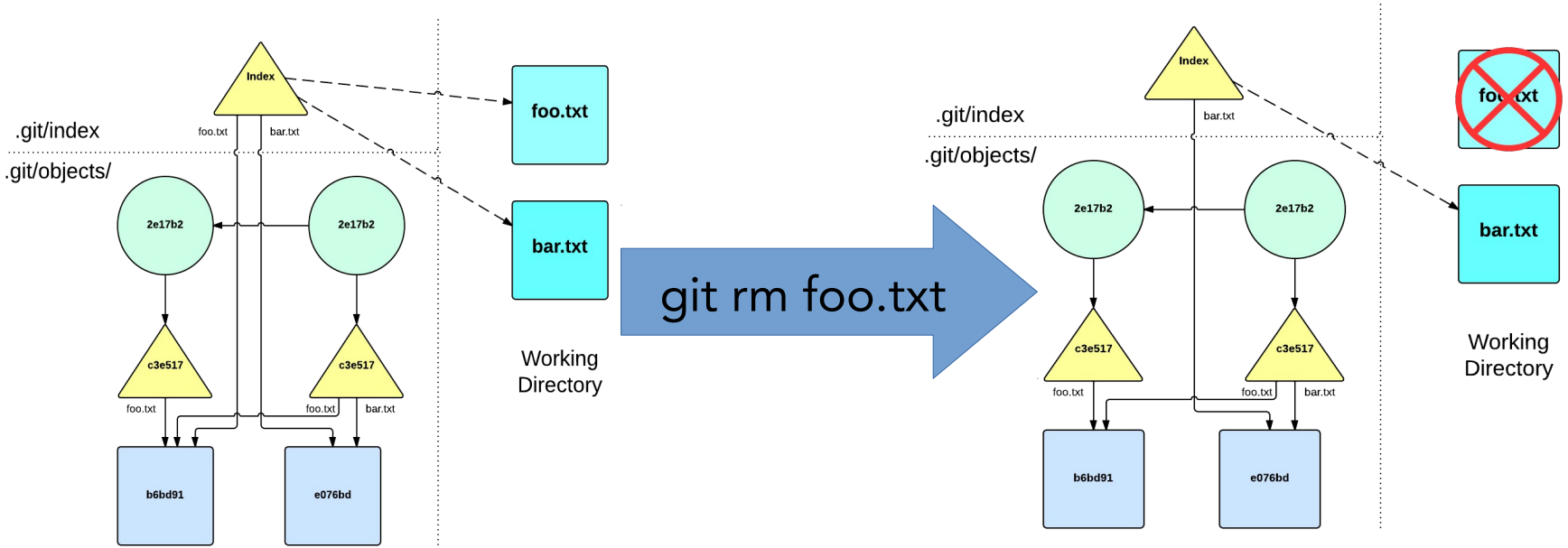


git rm

PERFORCE
Version everything.

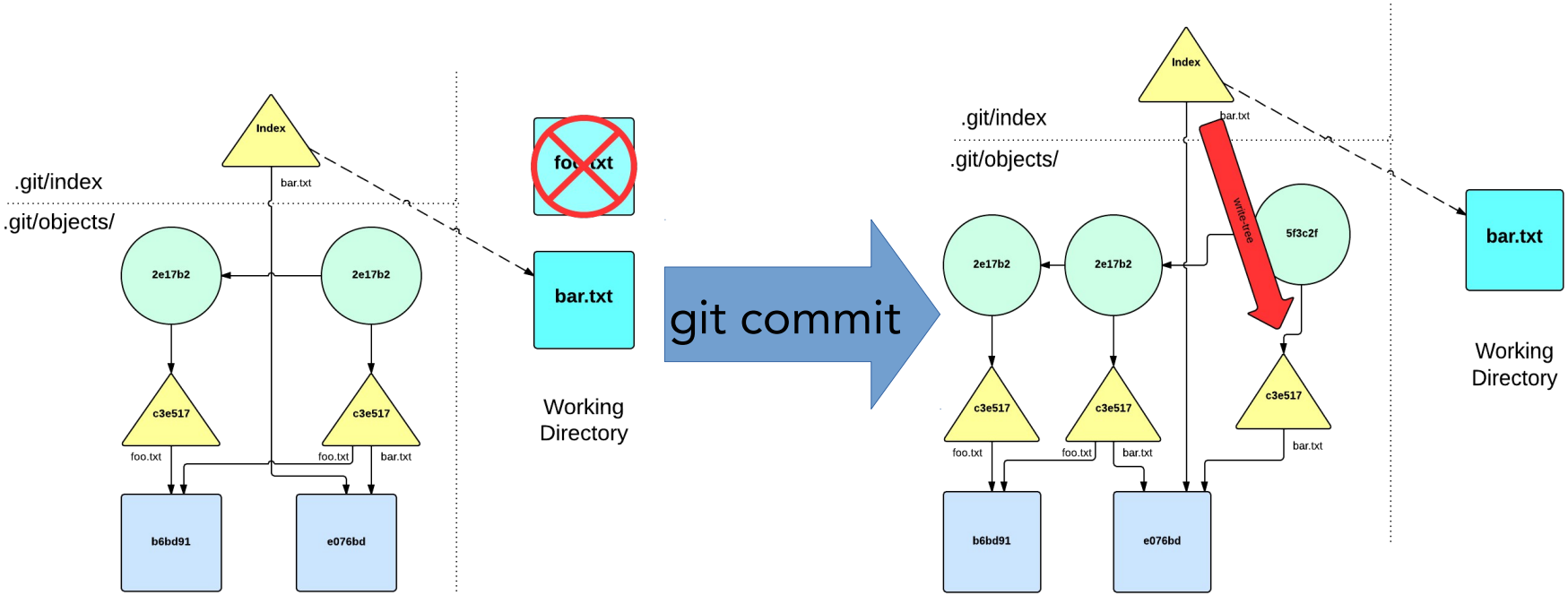
- Just like git add adds files to the index, git rm removes them, and removes the working copy as well.
- Pre-existing files in the object store are not touched.
- 'Removal' happens upon commit, when a tree lacking those files is stored in the git repo upon commit.
- Again, a graph to the rescue!

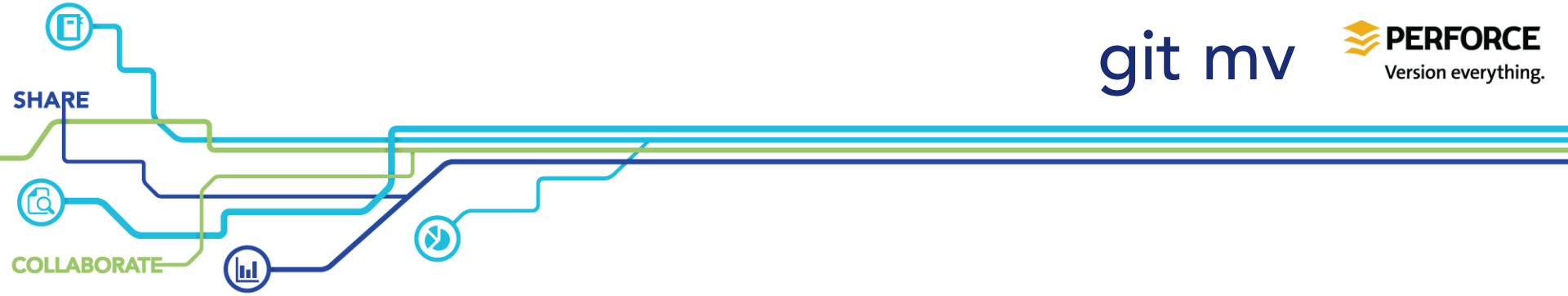
git rm


Index
.git/index
.git/objects/
foo.txt
bar.txt
2e17b2
2e17b2
c3e517
c3e517
foo.txt
foo.txt
bar.txt
b6bd91
e076bd
foo.txt
bar.txt
Working Directory
git rm foo.txt
Index
.git/index
.git/objects/
bar.txt
2e17b2
2e17b2
c3e517
c3e517
foo.txt
foo.txt
bar.txt
b6bd91
e076bd
bar.txt
Working Directory

git rm

PERFORCE
Version everything.





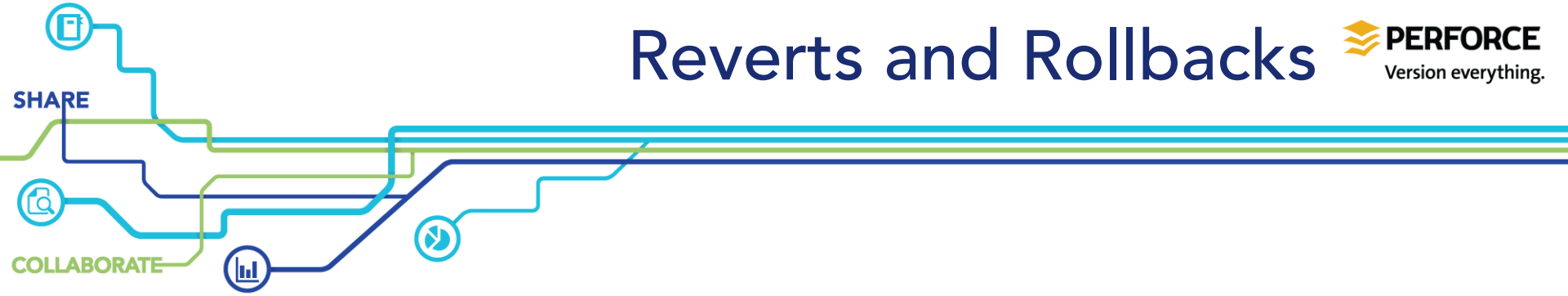
git mv

PERFORCE
Version everything.

- The git mv command moves or renames a file:

```
git mv <old_file> <new_file>
git mv <old_file> <new_dir>
```

- When renaming or moving the file, the index will point to the same blob (content doesn't change) but using the new name. During commit, the new tree will get written with the new file, but pointing to the old blob.



Reverts and Rollbacks

PERFORCE
Version everything.

- So now you know how to add, remove and move files around in a git repository.
- What happens, though, when you mess up the index prior to commit? How do you fix it?
- What can you do if mess up royally and actually commit the mess?

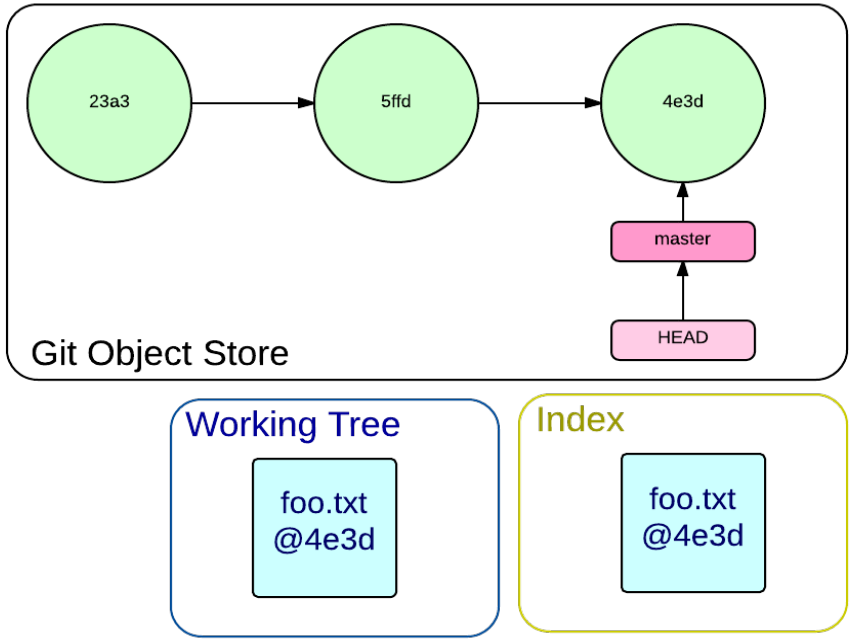
git reset

PERFORCE
Version everything.

- You can use git reset to undo and roll back bad commits.
- Git reset is a chimera, coming in two forms. New users can easily be confused by it. Worse, the first form has side-effects that are easy to miss or misunderstand.
- To truly master git reset, you need to understand how both forms work, and leverage the side-effects to your benefit.

git reset

PERFORCE
Version everything.



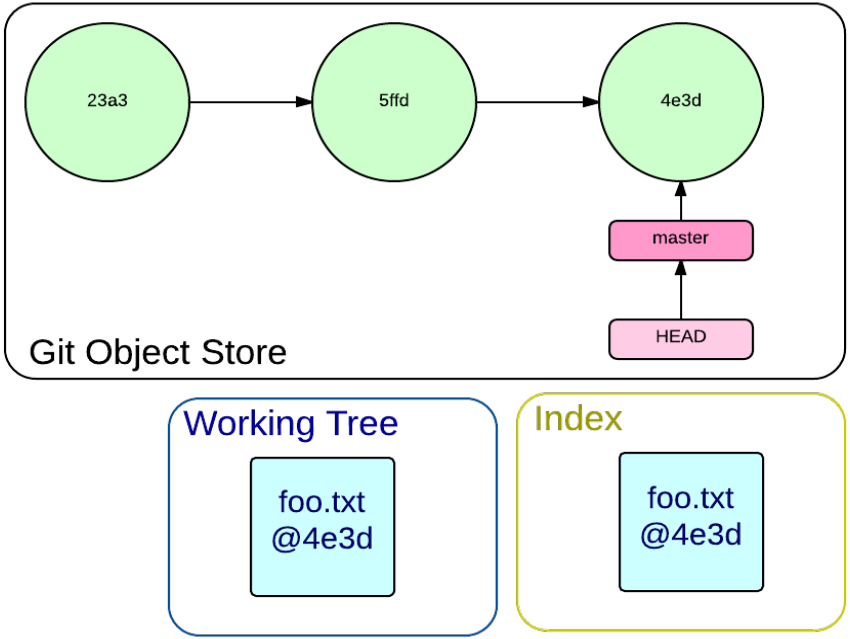
- A Refresher
- Git has three 'trees'*

- The working tree, which contains the sandbox
- The index, which contains the proposed commit
- The current commit, pointed to by HEAD

git reset

PERFORCE
Version everything.

Git reset [option] <tree-ish>



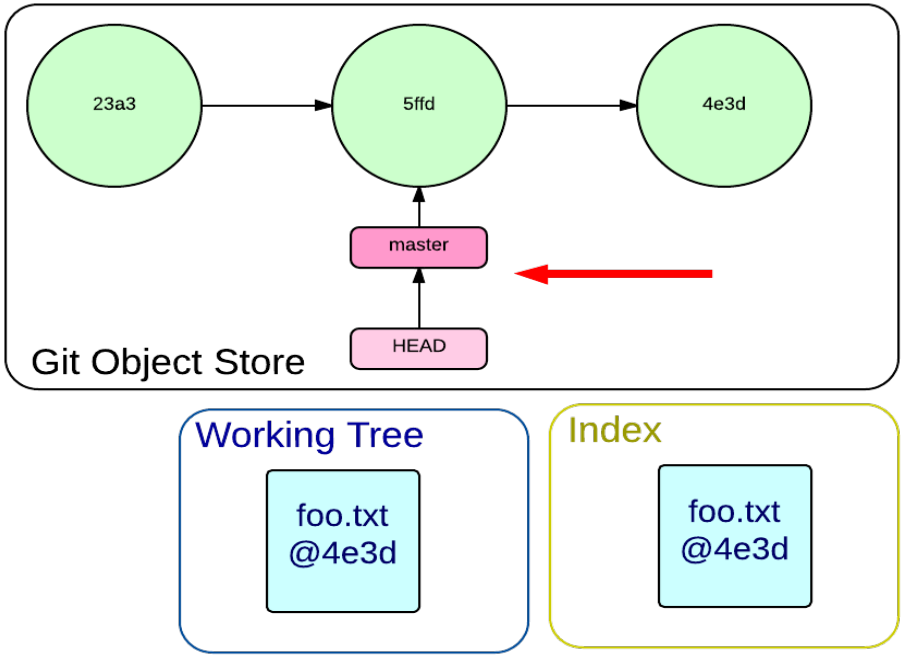
- The first form of git reset can affect all three trees
- This form is used to repoint the current HEAD to a new <tree-ish>, which (remember) can be a ref, commit or raw tree
- This can have various side-effects depending on the [option] used

git reset --soft

PERFORCE
Version everything.

SHARE
COLLABORATE

Git reset --soft HEAD~ *



- Points HEAD to the new commit, in this case the commit prior to HEAD
- The working tree and the index are unaffected.
- Because the index still contains the former HEAD contents, all files are ready for re-commit.

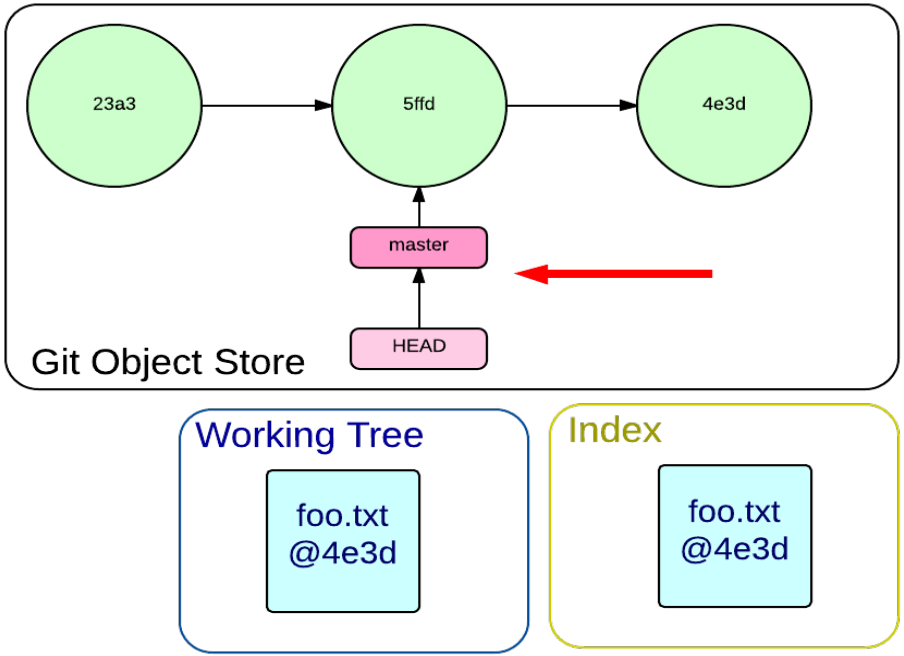
* Remember, HEAD~ is the commit prior to HEAD

git reset --soft

PERFORCE
Version everything.

SHARE
COLLABORATE

Git reset --soft HEAD~



- This means that if you were to issue a git commit, you would effectively re-create the former commit*
- This makes git reset –soft HEAD~ a sort of 'commit undo'

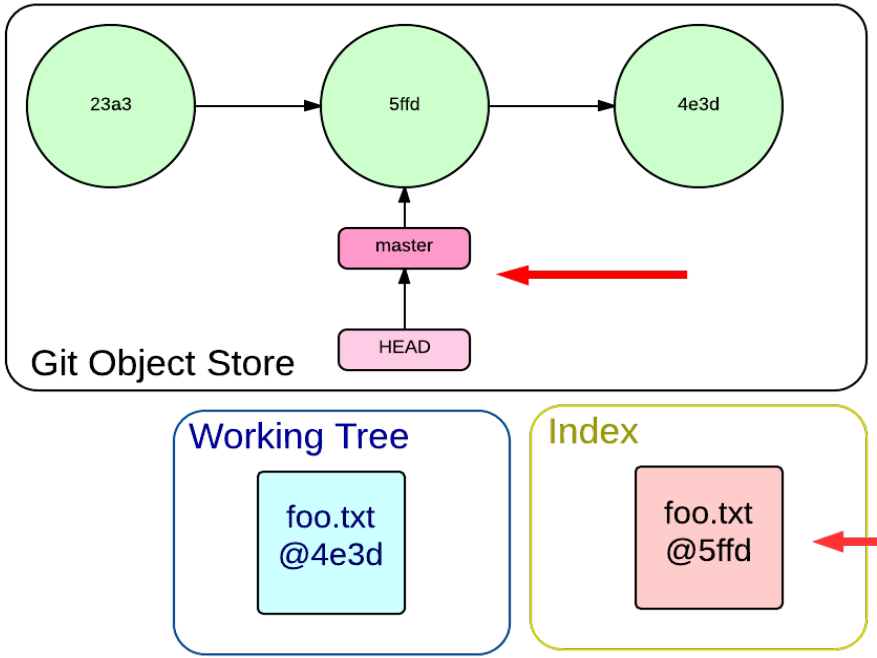
* not exactly – because the timestamp would Be different the hash of the new commit Wouldn't match the old one. But the root tree would be identical.

git reset --mixed

PERFORCE
Version everything.

SHARE
COLLABORATE

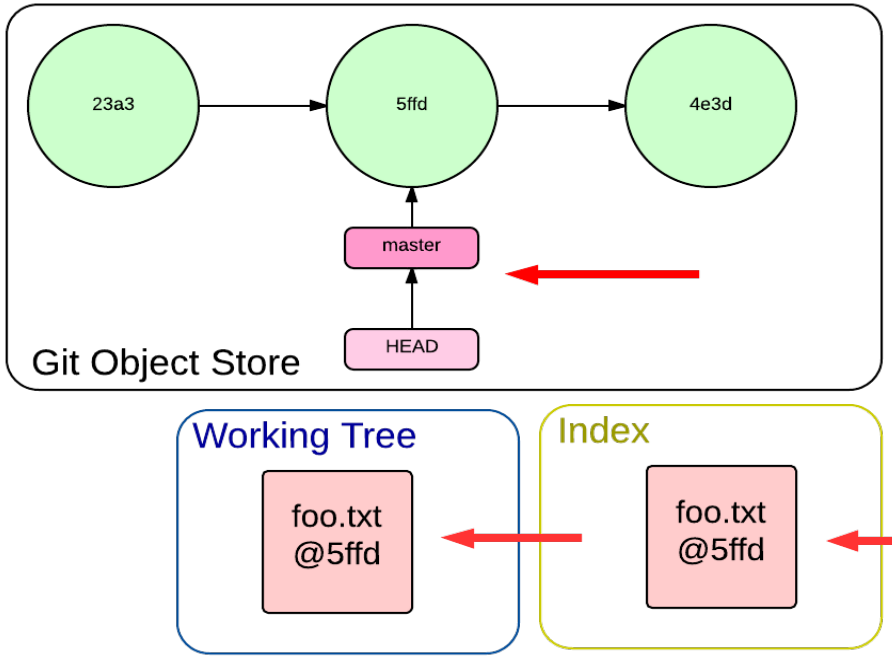
Git reset HEAD~



- There is an option implied here, `--mixed`. Because --mixed is the default option, it's omitted.
- HEAD is pushed back to the previous commit, like with --soft
- In addition, the index is changed to match the new commit.
- This has the effect of unstaging all the work done needed to produce the former HEAD, 4e3d.

git reset --hard

Git reset --hard HEAD~



- With --hard, not only is the HEAD reset to the new commit, but both the index and the working tree are modified to match the new HEAD.
- This makes it appear that you have just committed HEAD~ (5ffd).
- Use --hard whenever you want to reset your entire environment(all three trees) to a specific commit

git reset

PERFORCE
Version everything.

- Reset's heuristic can be broken down like this:

1. Move whatever HEAD points to, STOP if --soft
2. THEN, make the index look like that new commit, UNLESS --hard is specified
3. THEN, make the working tree look like the new commit.

git reset <file(s)>

PERFORCE
Version everything.

- We're not done with reset yet.
- Let's look at reset's second form:
  `git reset <file(s)>`
- This works differently than the other form, because you are can't reset HEAD to part of a commit.
- Therefore this form alters your index only, and is useful for undoing mistakes made during staging.

git reset <file(s)>

PERFORCE
Version everything.

```
➜  reset-example git:(master) ✗ git st
## master
A  .project
➜  reset-example git:(master) ✗
```

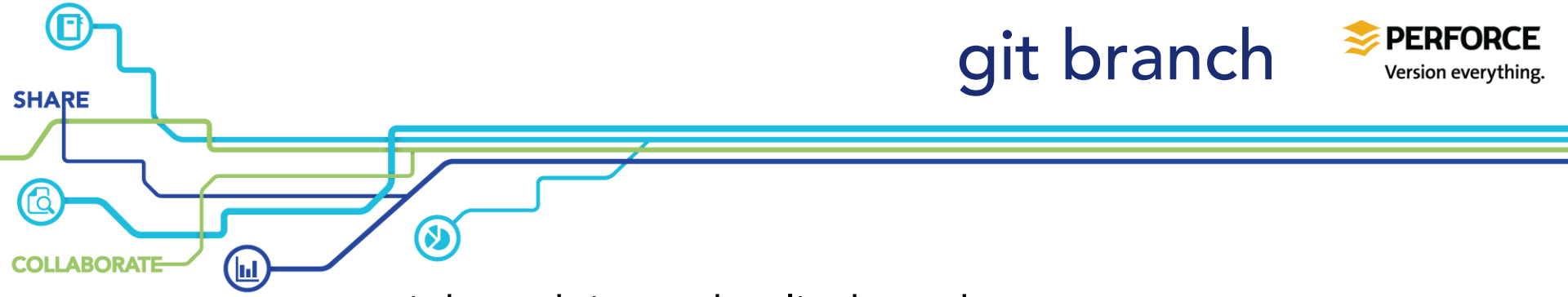
- Let's assume that we added a file, `.project`, to the index by mistake. We now want to remove it

git reset <file(s)>

PERFORCE
Version everything.

```
➜  reset-example git:(master) ✗ git reset .project
➜  reset-example git:(master) ✗ git st
## master
?? .project
➜  reset-example git:(master) ✗
```

- What this does is copy the version of <file(s)> that exists in HEAD into the index
- Practically, this results in unstaging the file, as the index contained the HEAD version prior to staging.



git branch

PERFORCE
Version everything.

- git branch is used to list branches, create new ones and delete them.
- By itself, git branch just lists the current branches in the repo.

```
➜  branch-example git:(master) git branch
  cow
* master
  sheep
➜  branch-example git:(master)
```

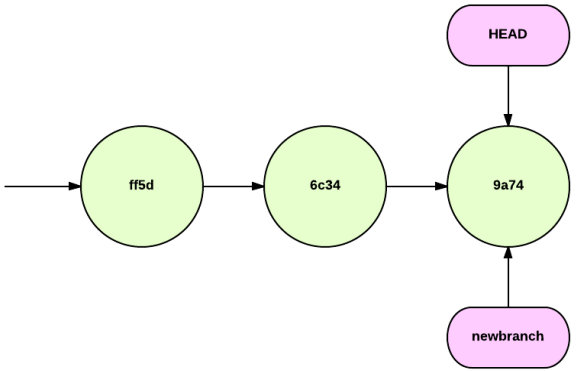
git branch

PERFORCE
Version everything.

SHARE

COLLABORATE

```
➜  git_presentation git:(master) ✗ git branch newbranch
➜  git_presentation git:(master) ✗ git rev-parse newbranch
9a7424a4d34e85dd0a9497cc30c4c06d2f79f21f
➜  git_presentation git:(master) ✗ git rev-parse HEAD
9a7424a4d34e85dd0a9497cc30c4c06d2f79f21f
➜  git_presentation git:(master) ✗
```



- The creation and destruction of branches is very simple. Remember when I said that branches were just refs?
- When you issue a `git branch <branchname>`, the following happens:
  1. A new ref, called a branch head, is created and named <branchname>
  2. The new branch head points to the current HEAD (the commit you are on).
- That's it. The real work is done when a commit is made on the branch.

git checkout

PERFORCE
Version everything.

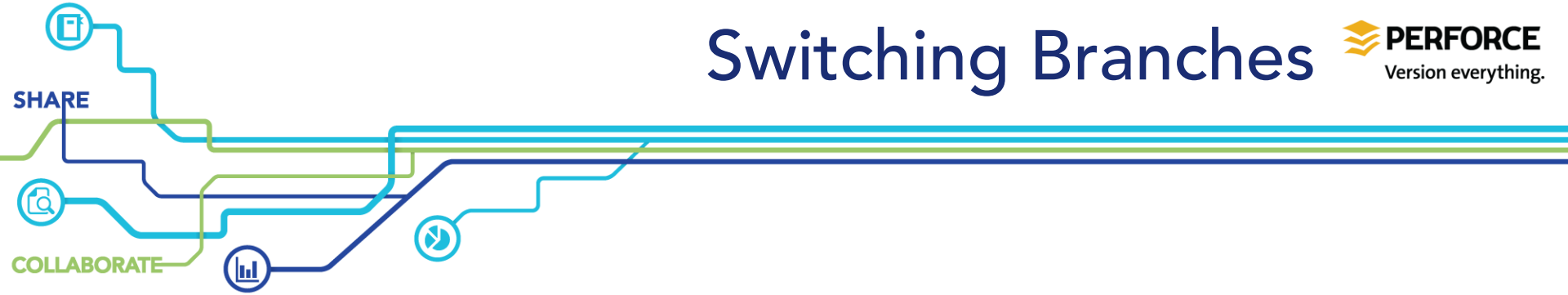
- Once we've created a branch, how do we use it?
- Git checkout allows you to switch to a branch with `git checkout <branch>`
- What does this mean?

```
➜  branch-example git:(master) git branch
  cow
* master
  sheep
➜  branch-example git:(master) git checkout cow
Switched to branch 'cow'
➜  branch-example git:(cow) git branch
* cow
  master
  sheep
➜  branch-example git:(cow)
```

git checkout

PERFORCE
Version everything.

- Like many commands in git, git checkout is more general than it appears, and does different things depending on how it is invoked.
- It is used most commonly for two purposes:
    - Switch to another branch
    - Update the working tree from another branch, tag or commit



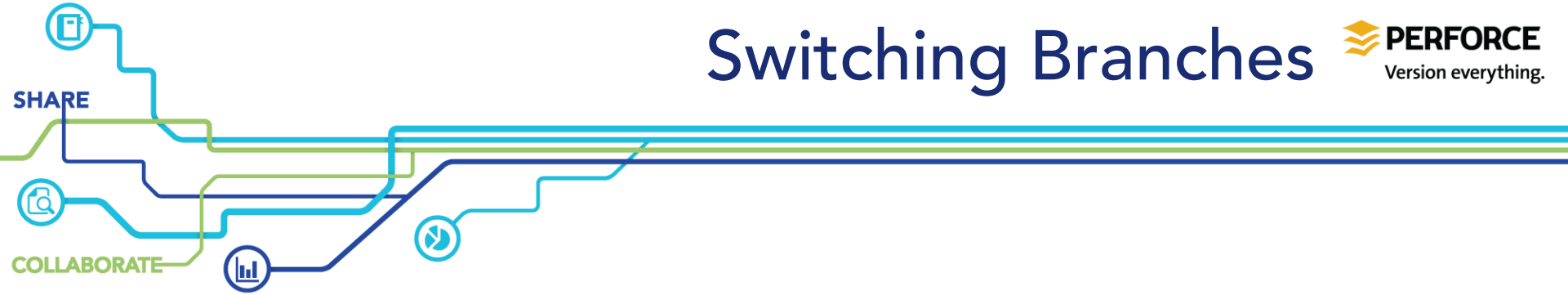
Switching Branches

PERFORCE
Version everything.

- Git checkout <branch> prepares the git repo for working on <branch> by doing the following:

  1. Update the files in the working tree to match those stored by the commit at the branch head*
  2. Update the index to make it current with the working tree.
  3. Switch the HEAD ref so that it points to the branch head.

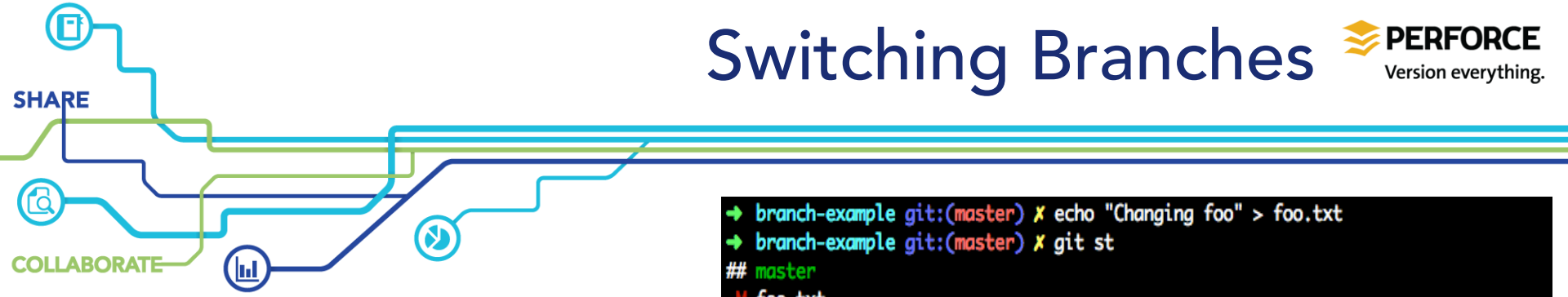
* Recall that the working tree is the union of the files from the last commit + any local changes to those files.



Switching Branches

PERFORCE
Version everything.

- Because switching branches updates the working tree to match <branch>, if you have any changes in the working tree (staged or otherwise) that would be overwritten, git will refuse to switch branches until you either stash or commit your work.
- If local changes would not be overwritten, then they are moved along with the branch switch.



Switching Branches

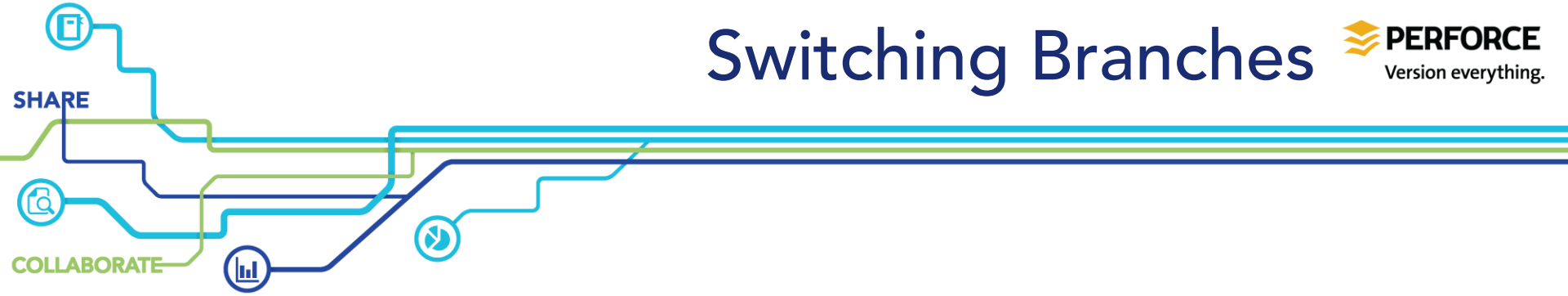
PERFORCE
Version everything.

Fails because foo.txt is present in the commit pointed to by cow

```
➜ branch-example git:(master) ✗ echo "Changing foo" > foo.txt
➜ branch-example git:(master) ✗ git st
## master
 M foo.txt
➜ branch-example git:(master) ✗ git checkout cow
error: Your local changes to the following files would be overwritten by checkout:
        foo.txt
Please, commit your changes or stash them before you can switch branches.
Aborting
➜ branch-example git:(master) ✗
```

```
➜ branch-example git:(master) ✗ git add baz.txt
➜ branch-example git:(master) ✗ git st
## master
A  baz.txt
➜ branch-example git:(master) ✗ git checkout cow
A       baz.txt
Switched to branch 'cow'
➜ branch-example git:(cow) ✗ git st
## cow
A  baz.txt
➜ branch-example git:(cow) ✗
```

Succeeds because baz.txt is not present in the cow branch



Switching Branches

PERFORCE
Version everything.

- Because creating a new branch is such a simple operation, you can create and checkout a new branch with a single command:

```
git checkout -b <new_branch>
```

Updating the working tree

PERFORCE
Version everything.

- The other purpose of git checkout is to update parts of or the entire contents of your working tree with the contents of another commit.
- In this aspect, it takes the form
  `git checkout <tree-ish>* <file(s)>`
- If <tree-ish> is not specified, then it is the index.
- Useful for when you want to revert some files, but want to keep changes to others.

* Remember that <tree-ish> is anything that resolves to a tree – it could be a ref, a commit or an actual tree.

Updating the working tree

PERFORCE
Version everything.

Restoring a file to the version stored by the index (undoing an unstaged change)

```
➜ branch-example git:(master) echo "Destroy all humans\!" > bender.txt
➜ branch-example git:(master) ✗ git st
## master
 M bender.txt
➜ branch-example git:(master) ✗ git checkout bender.txt
➜ branch-example git:(master) git st
## master
➜ branch-example git:(master)
```

```
➜ branch-example git:(master) cat bender.txt
1. Bend things
2. Cheese it!
3. KILL ALL HUMANS
➜ branch-example git:(master) git checkout HEAD~ bender.txt
➜ branch-example git:(master) ✗ cat bender.txt
1. Bend things
2. Cheese it!
➜ branch-example git:(master) ✗
```

Checking out the previous version of a file

Updating the working tree

PERFORCE
Version everything.

- Checking out previous versions of a file or directory updates the file in your working tree as if you had modified them directly.

- Any rollback is a matter of checking out the files from the commit previous to the one where you made the mistakes, and then staging and committing.

Deleting a branch

PERFORCE
Version everything.

- Delete a branch with `git branch -d <branch>`
- If a branch has unmerged changes, it won't be deleted and you will be given a warning

```
➜ cow-sheep git:(master) git branch -d vapor
error: The branch 'vapor' is not fully merged.
If you are sure you want to delete it, run 'git branch -D vapor'.
➜ cow-sheep git:(master)
```

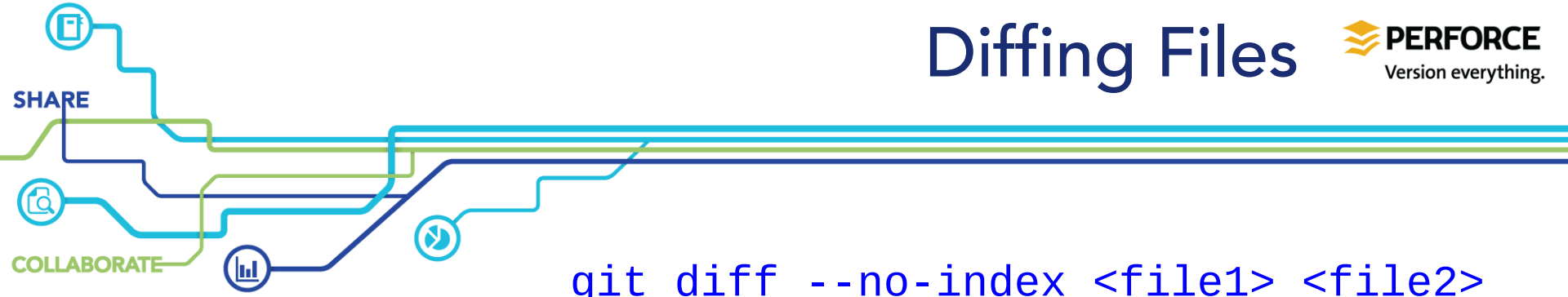
- You can override the warning with `-D`.
- If you do this, you will orphan the commits on that branch. Eventually they will be reaped by git's garbage collection routines.

Diffing Files

PERFORCE
Version everything.

- Git provides a diff command that can be used to compare the index to the working tree, the index and another tree/commit, between two blob objects, or between two files on disk.
- By default, `git diff` with no options will return the differences between the index and the working tree – that is, differences that you haven't staged yet.

```
➜ cow-sheep git:(wisecow) ls
foo.txt     wisecow.txt
➜ cow-sheep git:(wisecow) fortune | cowsay > wisecow.txt
➜ cow-sheep git:(wisecow) ✗ git diff
diff --git a/wisecow.txt b/wisecow.txt
index 183390e..ab3dea0 100644
--- a/wisecow.txt
+++ b/wisecow.txt
@@ -1,10 +1,9 @@
- _____________________________________
-/ There are three things men can do with \
-| women: love them, suffer for them, or  |
-| turn them into literature.             |
-|                                        |
-\ -- Stephen Stills                      /
- -------------------------------------
+ _____________________________________
+/ There never was a good war or a bad \
+| peace.                               |
+|                                      |
+\ -- B. Franklin                       /
+ -------------------------------------
         \  ^__^
          \ (oo)\_______
            (__)\       )\/\
➜ cow-sheep git:(wisecow) ✗ git st
## wisecow
 M wisecow.txt
➜ cow-sheep git:(wisecow) ✗
```



Diffing Files

PERFORCE
Version everything.

`git diff --no-index <file1> <file2>`

- You can diff between two files on disk (a la normal dif) by passing diff the `--no-index` option.
- The `-U<n>` flag is used to provide <n> lines of context around a diff.

```
➜ cow-sheep git:(wisecow) ✗ git diff -U6 --no-index quiet_sheep.txt loud_sheep.txt
diff --git a/quiet_sheep.txt b/loud_sheep.txt
index 292730e..4bf1a4e 100644
--- a/quiet_sheep.txt
+++ b/loud_sheep.txt
@@ -1,8 +1,8 @@
  ________
-< Baaaaaaa >
+< BAAAAAA! >
  ---------
   \
    \
     __
     UooU\.'@@@@@@`.
     \__/(@@@@@@@@@@)
➜ cow-sheep git:(wisecow) ✗
```

Diffing Files

PERFORCE
Version everything.

`git diff <tree-ish> <tree-ish>`

- You can diff between two different tree-ish objects (refs, tags, and commits)

```
➜ cow-sheep git:(wisecow) ✗ git diff cow sheep
diff --git a/foo.txt b/foo.txt
index cba8bcf..1dd2ac7 100644
--- a/foo.txt
+++ b/foo.txt
@@ -1,8 +1,11 @@
- _____________________________
-< Cow to Sheep, come in Sheep. >
- -----------------------------
-         \   ^__^
-          \  (oo)\_______
-             (__)\       )\/\
-                 ||----w |
-                 ||     ||
+ _____________________________
+< This is Sheep, over and out >
+ -----------------------------
+  \
+   \
+      __
+     UooU\.'@@@@@@`.
+     \__/(@@@@@@@@@@)
+          (@@@@@@@@)
+          `YY~~~~YY'
+           ||    ||
➜ cow-sheep git:(wisecow) ✗
```

Diffing Files

PERFORCE
Version everything.

- You can diff between the working tree and a given commit...

```
git diff <tree-ish>
```

```
➜  cow-sheep git:(master) echo "3. KILL ALL HUMANS" >> bender.txt
➜  cow-sheep git:(master) ✗ git diff master
diff --git a/bender.txt b/bender.txt
index 8432efe..ef31b7c 100644
--- a/bender.txt
+++ b/bender.txt
@@ -1,2 +1,3 @@
 1. Bend things
 2. Cheese it!
+3. KILL ALL HUMANS
➜  cow-sheep git:(master) ✗
```

Diffing Files

PERFORCE
Version everything.

- Or between the index and a given commit

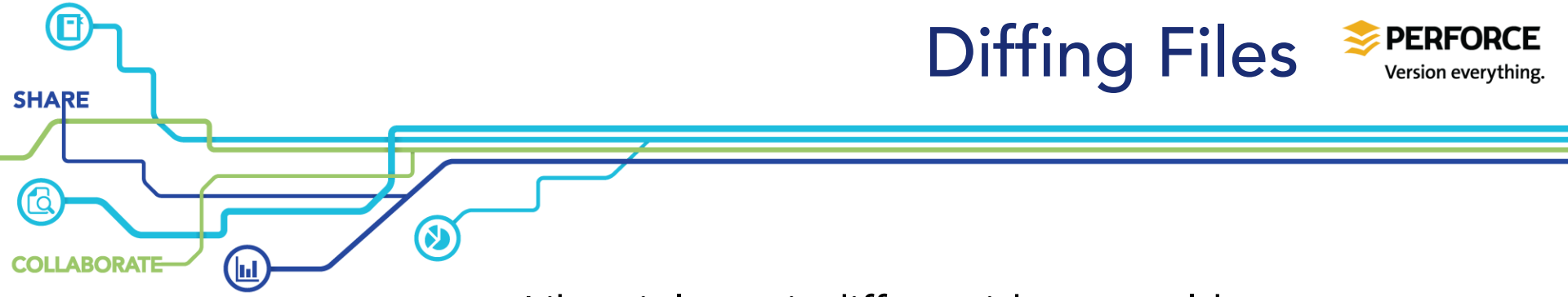
```
git diff --cached <tree-ish>
```

```
➜ cow-sheep git:(master) ✗ git st
## master
 M bender.txt
M  professor.txt
➜ cow-sheep git:(master) ✗ git diff --cached master
diff --git a/professor.txt b/professor.txt
index b0896a9..5b6968c 100644
--- a/professor.txt
+++ b/professor.txt
@@ -1 +1,2 @@
 Good news, everyone!
+Tomorrow you'll be delivering a crate of subpoenas to Sicily 8, the Mob Planet!
➜ cow-sheep git:(master) ✗
```

- You can diff between the working tree and a given commit...

```
git diff <tree-ish>
```

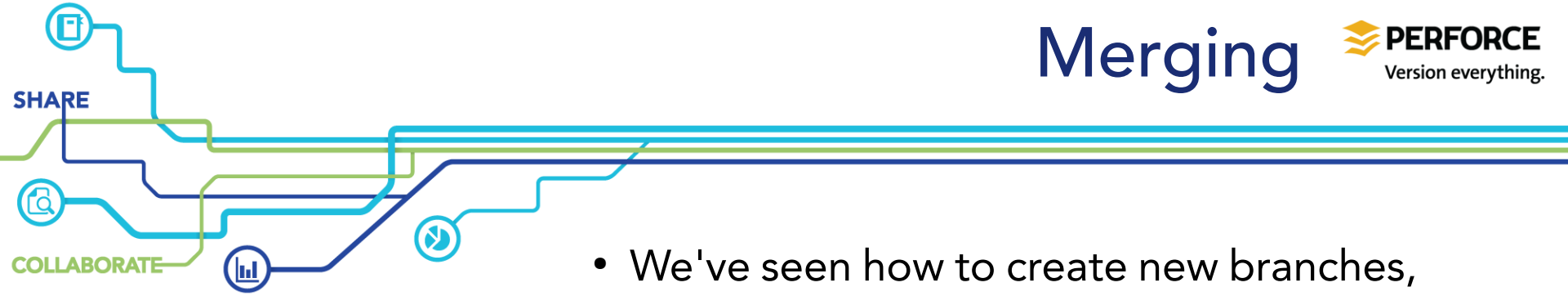
```
➜ cow-sheep git:(master) echo "3. KILL ALL HUMANS" >> bender.txt
➜ cow-sheep git:(master) ✗ git diff master
diff --git a/bender.txt b/bender.txt
index 8432efe..ef31b7c 100644
--- a/bender.txt
+++ b/bender.txt
@@ -1,2 +1,3 @@
 1. Bend things
 2. Cheese it!
+3. KILL ALL HUMANS
➜ cow-sheep git:(master) ✗
```



Diffing Files

PERFORCE
Version everything.

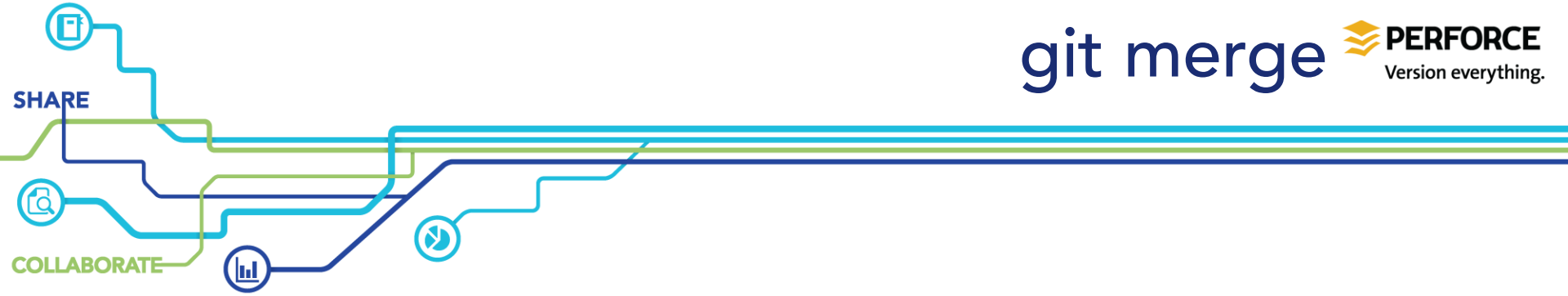
- Like git log, git diff provides a wealth of options
- One of these, `-S` (aka *Pickaxe*) is common to both git log and git diff, and allows you to track changes in a specific block of code (not just a file).
- `git help diff` – go read it.



Merging

PERFORCE
Version everything.

- We've seen how to create new branches, and create new commits on those branches.
- Now how do we reconcile changes from one branch to another?
- We have two options – we can rebase the commits on the other branch, or we can merge them.
- Let's look at merging first.



git merge

PERFORCE
Version everything.

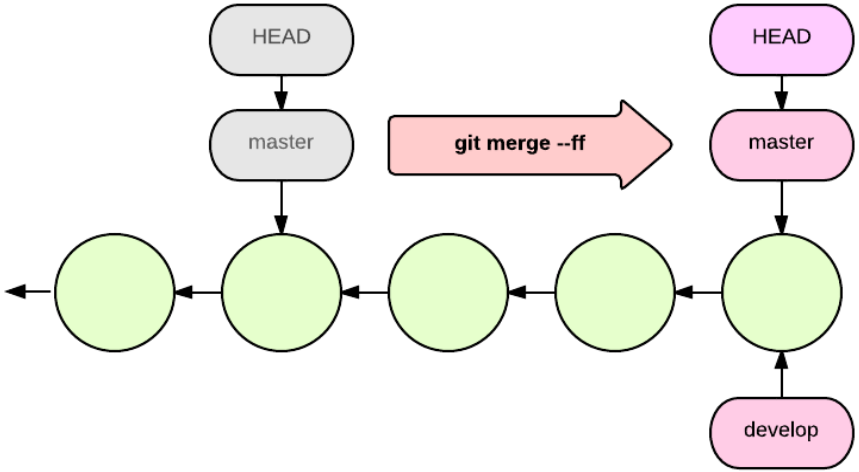
- Use git merge to incorporate changes from the named tree-ish object (usually a branch head) to your current branch.
- `git merge <branch>`
- When doing a merge, git will attempt, one at a time, three strategies to fold the changes from <branch> into your current branch.

Fast-forward merging

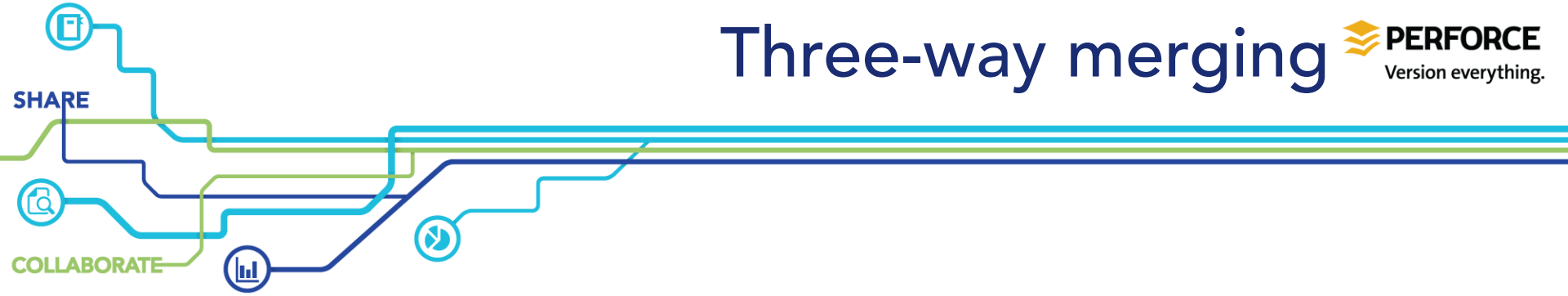
PERFORCE
Version everything.

SHARE
COLLABORATE

- First git will attempt a fast-forward merge. This is the simplest kind, and does not result in a merge commit.



- When all the commits on the branch to merge are upstream of HEAD, git will simply move the current branch ref to point to the head ref of the merged branch, and update the working and index trees to match the new commit.

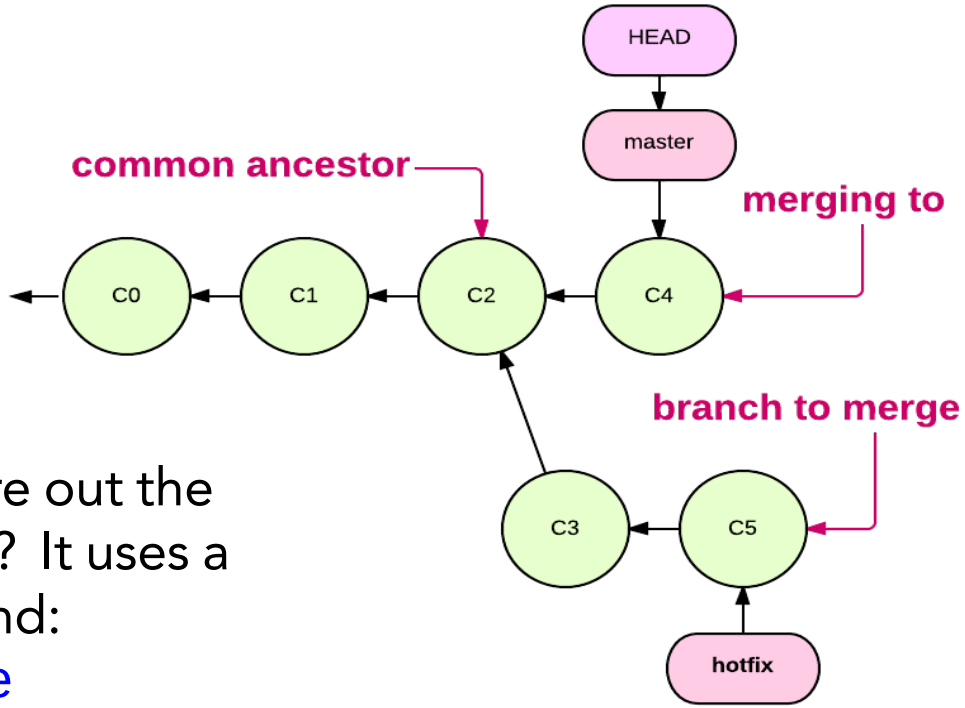


Three-way merging

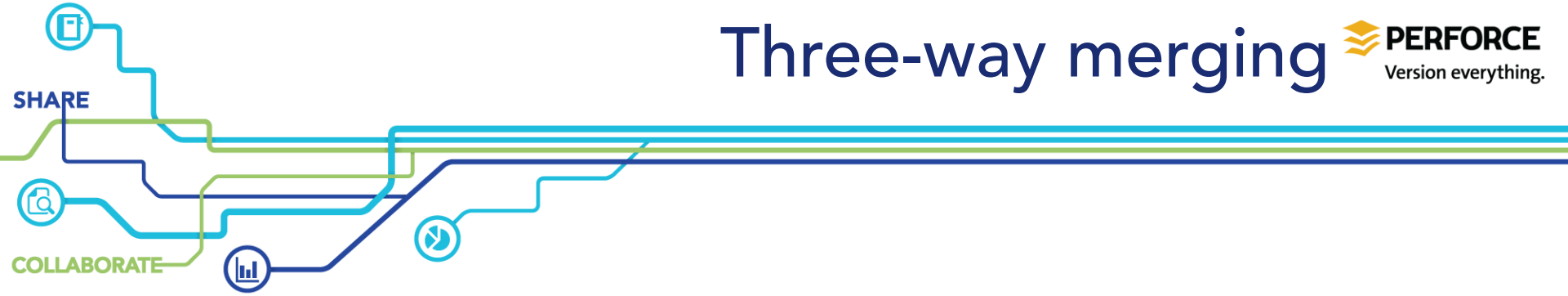
PERFORCE
Version everything.

- However, if the history of the current branch has diverged from that of the branch to merge, then simply fast-forwarding the HEAD will not work.
- In this case git will attempt a three-way merge using the branch head of the branch to merge, the current HEAD, and a common ancestor.
- It will first attempt the simpler of the remaining two merge strategies, called the recursive strategy.

Three-way merging



- How does git figure out the common ancestor? It uses a plumbing command:
  `git merge-base`



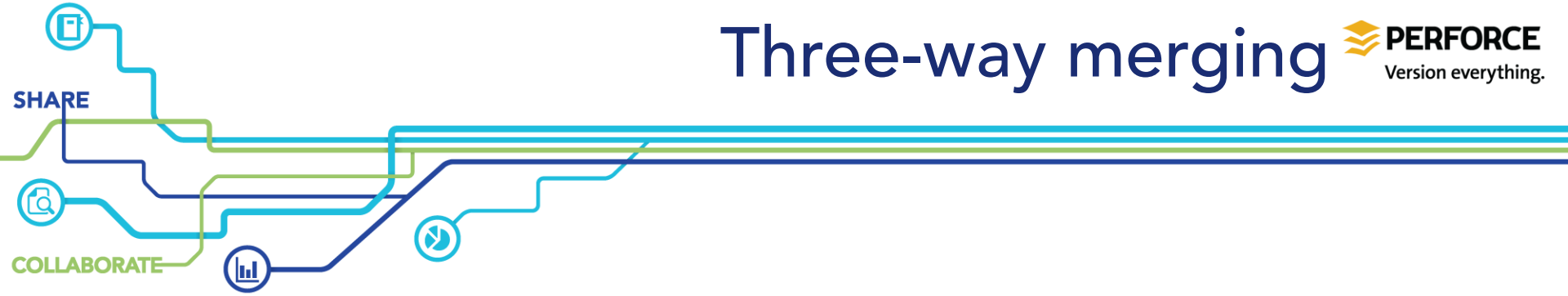
Three-way merging

PERFORCE
Version everything.

- You can see what it decides by calling the command directly: `git merge-base master hotfix`

```
➜  git-examples git:(master) ✗ git merge-base master hotfix
40355164cd645505de722826a816c380d08a0390
➜  git-examples git:(master) ✗ git lg
* db3ceb8 - (HEAD, master)  C4 (58 seconds ago) <Jan Van Uytven>
* 4035516 -  C2 (2 minutes ago) <Jan Van Uytven>
* c480f0a -  C1 (2 minutes ago) <Jan Van Uytven>
* 90a51ef -  C0 (2 minutes ago) <Jan Van Uytven>
```

- Here it decided that commit 403551, aka 'C2', was the best merge base.

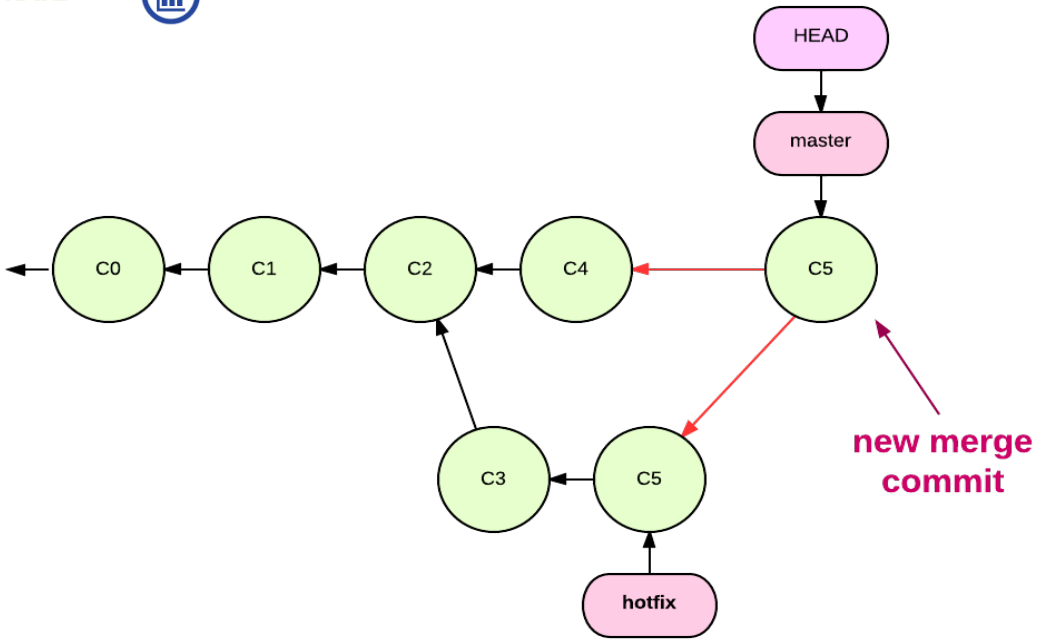


Three-way merging

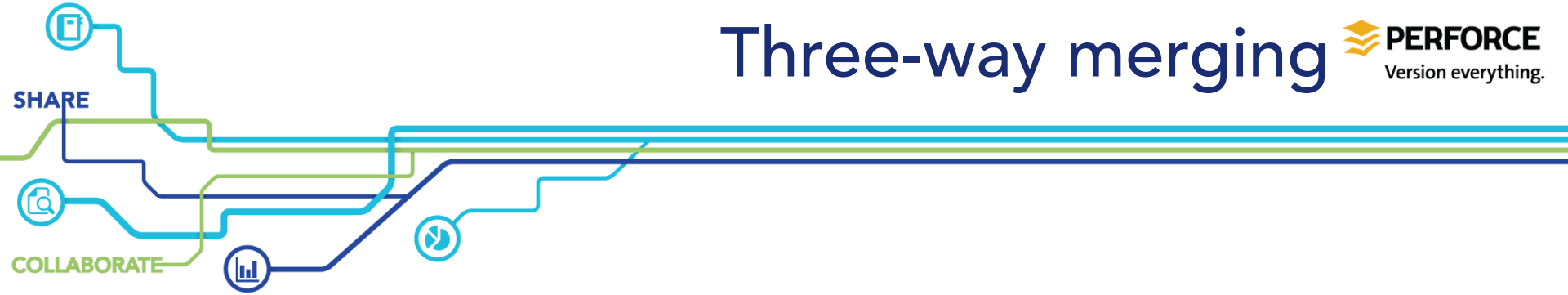
PERFORCE
Version everything.

- Assuming there are no conflicts, git will create a new commit to hold the merged tree and prompt the user for a commit message.
- The branch ref for master will then be moved to the new merge commit, and the working tree and index will be updated with the results of the merge.

Three-way merging



- Note that git merge doesn't stage anything to the index. If it can, it will create the merge commit.
- If it can't, then it reports a conflict.



Three-way merging

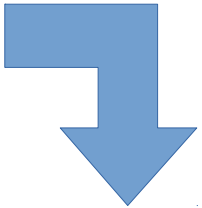
PERFORCE
Version everything.

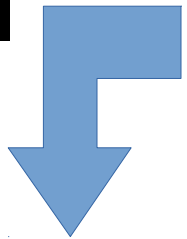
- A conflict requires user intervention to resolve the files
- Git will place markers in the conflict files
- You can use `git status` to reveal which files have conflicts.
- Conflicts are resolved by manually editing the files, removing the markers and then adding them to the index.
- Once all conflicts are resolved, you do a `git commit` to complete the merge.

Three-way merging

PERFORCE
Version everything.

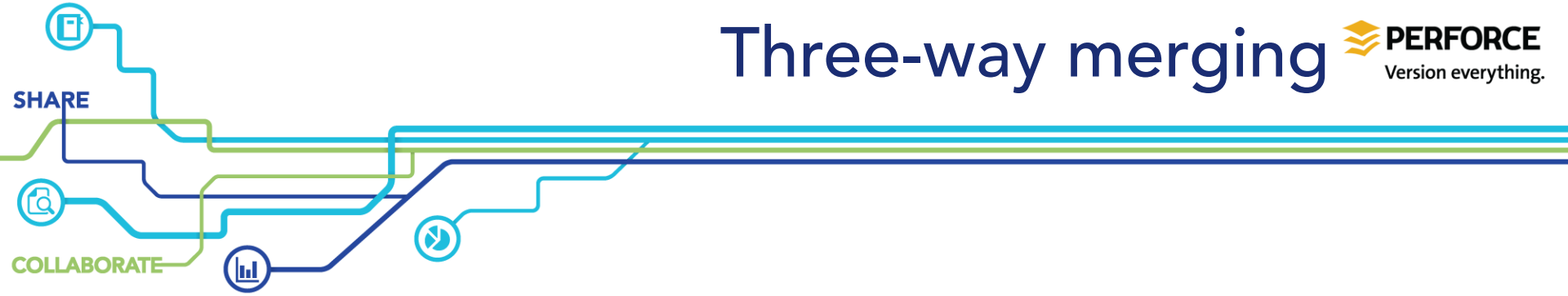
```
➜  git-examples git:(master) ✗ git merge hotfix
Auto-merging foo.txt
CONFLICT (content): Merge conflict in foo.txt
Automatic merge failed; fix conflicts and then commit the result.
➜  git-examples git:(master) ✗ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:   foo.txt
```





```
➜  git-examples git:(master) ✗ git add foo.txt
➜  git-examples git:(master) ✗ git st
## master
M  foo.txt
```

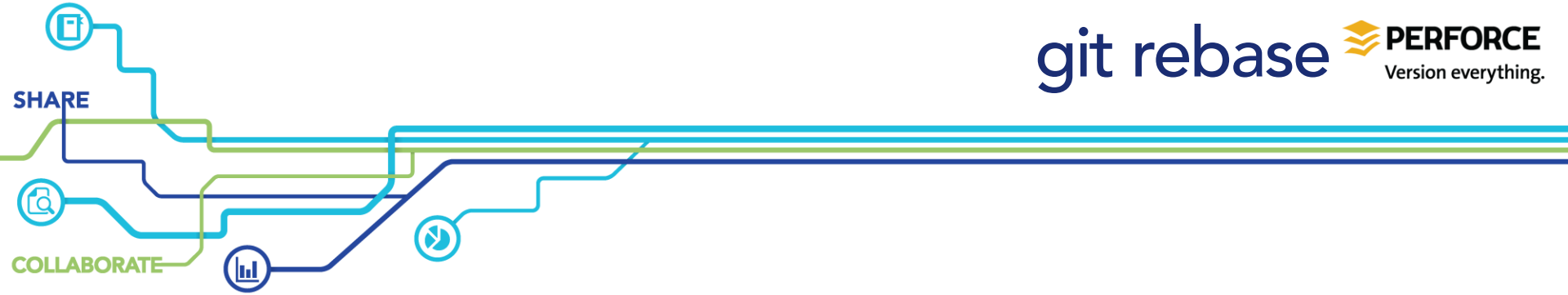
```
➜  git-examples git:(master) ✗ git commit
[master fb1b6f0] Merge branch 'hotfix'
➜  git-examples git:(master) ✗ git lg
*   fb1b6f0 - (HEAD, master)  Merge branch 'hotfix' (8 seconds ago) <Jan Van Uytven>
|\
| * 0f59f90 - (hotfix)  C5 (24 minutes ago) <Jan Van Uytven>
```



Three-way merging

PERFORCE
Version everything.

- Git merges can be done with multiple branches:
  `git merge hotfix task-1 zoidberg-want-banana`
- When there are multiple branches, git resorts to the third merge strategy, called the 'octopus' strategy.
- Practically, this works the same as the three-way merge



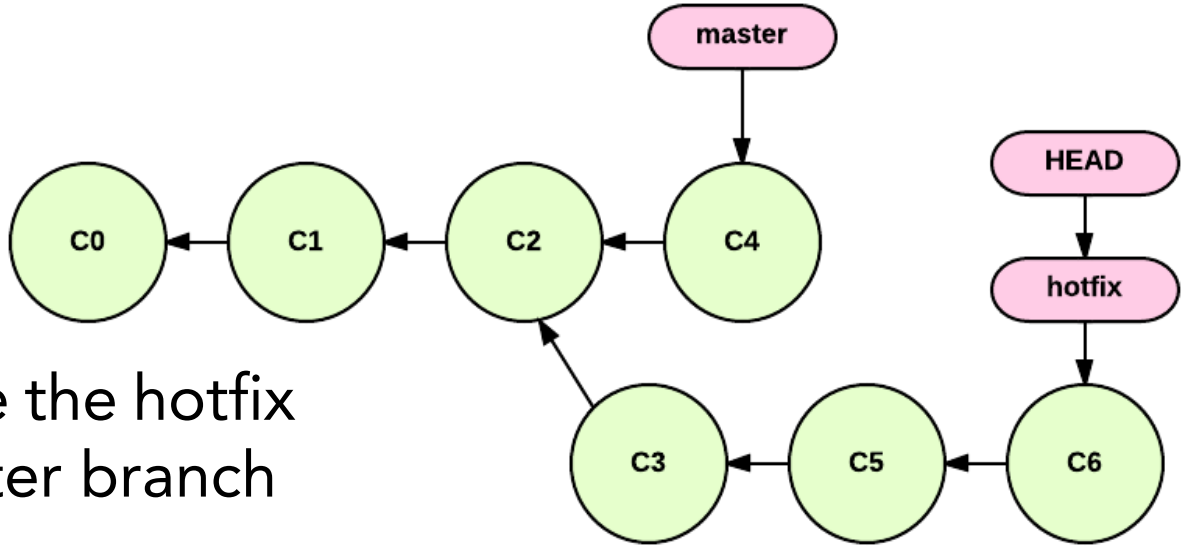
git rebase

PERFORCE
Version everything.

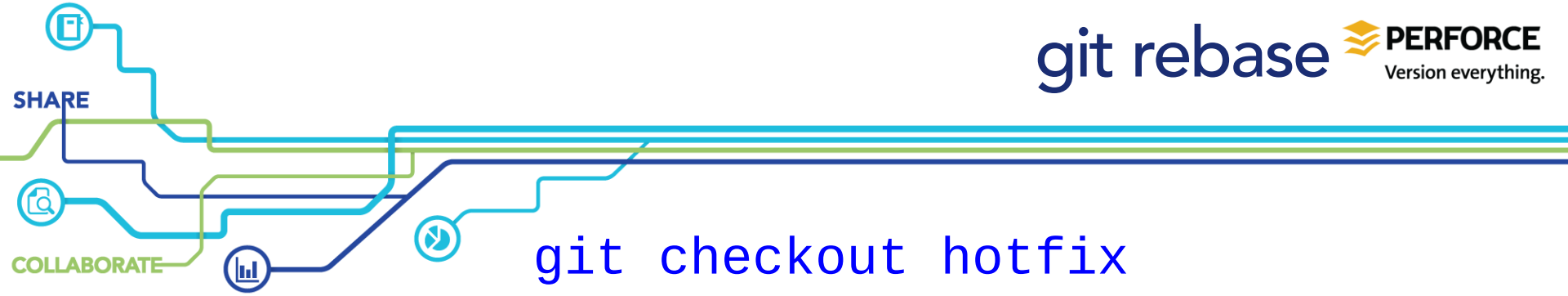
- The other option for integrating branches is `git rebase`
- It allows you to cleanly replay the history of a branch onto your current branch, making it look like the work was done in serial instead of in parallel
- This is great for maintaining a single, clean codeline (usually master) that can be easily rebased onto and applied.
- Let's looks at an example.

git rebase

PERFORCE
Version everything.



- We want to integrate the hotfix branch into the master branch
  - Instead of merging it, we're going to replay the hotfix branch onto master.



git rebase

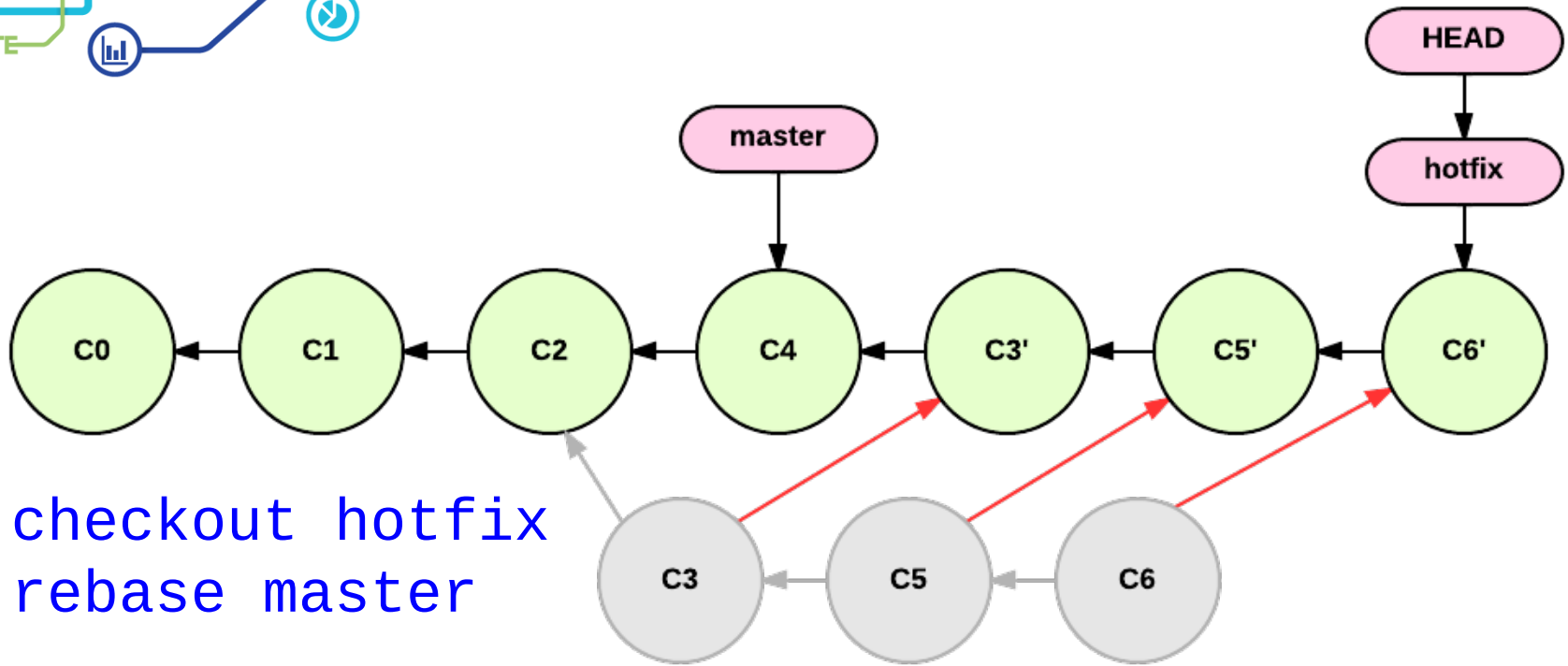
PERFORCE
Version everything.

```
git checkout hotfix
git rebase master
```

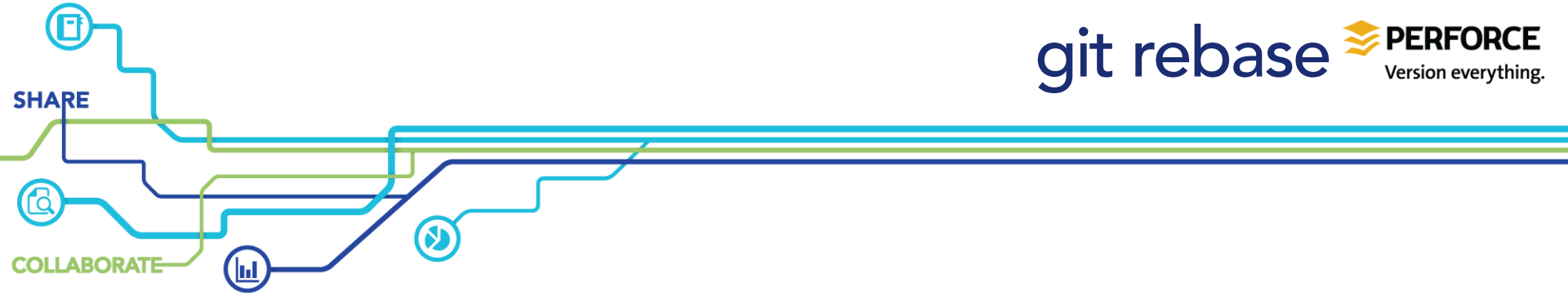
- Start by taking the first commit of hotfix and replaying it on the tip of master
- Do the same with the second and third commits of hotfix.
- Each time rebase replays a commit, it creates a new one that contains the same information but has a different parent.

git rebase

PERFORCE
Version everything.




```
git checkout hotfix
git rebase master
```



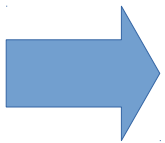
git rebase

PERFORCE
Version everything.

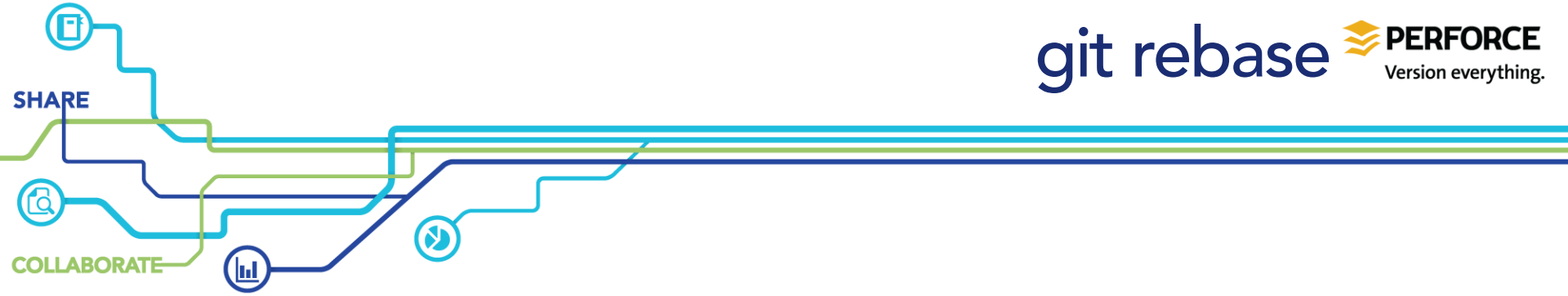
- At this point, we have flattened hotfix onto master
- All we need to do now is fast-forward master
- No merge commits are required

git rebase


```
➜  rebase-example git:(hotfix) git lg hotfix master
* adcd56c - (HEAD, hotfix)  C6 (7 minutes ago) <ysgard>
* 23e7e23 -  C5 (8 minutes ago) <ysgard>
* 0a33f49 -  C3 (8 minutes ago) <ysgard>
| * 8959c17 - (master)  C4 (8 minutes ago) <ysgard>
|/
* a7ecf59 -  C2 (9 minutes ago) <ysgard>
* c112a7a -  C1 (9 minutes ago) <ysgard>
* c2a9227 -  C0 (10 minutes ago) <ysgard>
➜  rebase-example git:(hotfix)
```

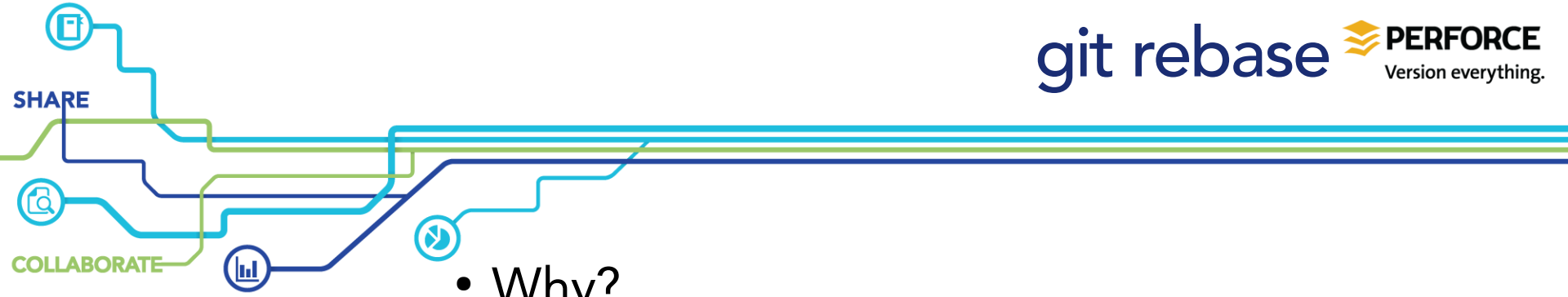


```
➜  rebase-example git:(hotfix) git rebase master
First, rewinding head to replay your work on top of it...
Applying: C3
Applying: C5
Applying: C6
➜  rebase-example git:(hotfix) git lg hotfix master
* c4e2bb4 - (HEAD, hotfix)  C6 (9 minutes ago) <ysgard>
* 1e0448d -  C5 (9 minutes ago) <ysgard>
* 16111ea -  C3 (9 minutes ago) <ysgard>
* 8959c17 - (master)  C4 (23 minutes ago) <ysgard>
* a7ecf59 -  C2 (24 minutes ago) <ysgard>
* c112a7a -  C1 (24 minutes ago) <ysgard>
* c2a9227 -  C0 (24 minutes ago) <ysgard>
➜  rebase-example git:(hotfix)
```



- Git rebase is powerful. You can not only flatten branches, you can also reposition them using rebase's `--onto` option.
- but comes with a massive caveat:

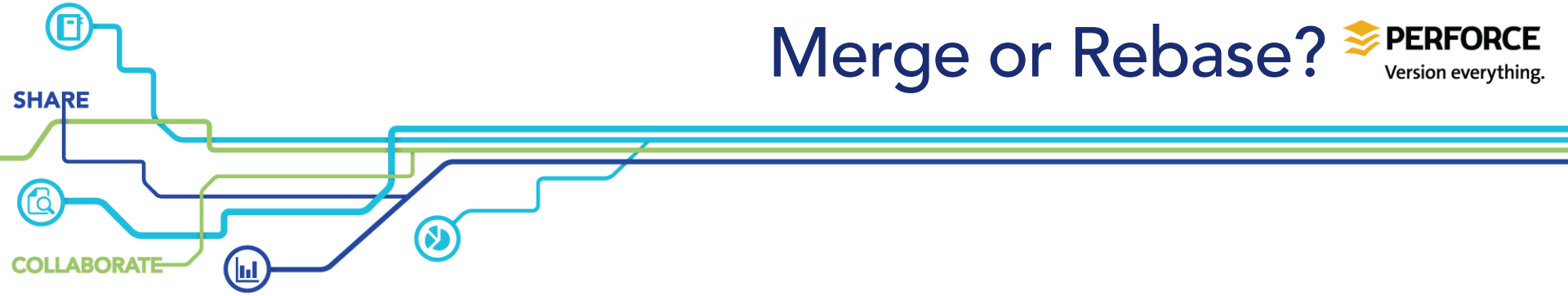
Don't rebase commits that you have already pushed!



git rebase

PERFORCE
Version everything.

- Why?
- Because when you git rebase, you're dropping and re-ordering commits.
- If those commits already exist in a remote repo, then other developers are going to have copies of those commits.
- Which means the next time you push your changes, you're going to condemn them to rebase hell.



Merge or Rebase?

PERFORCE
Version everything.

- So which should you use, merge or rebase?
- It depends on what you're trying to accomplish.

- Merging makes it clear that work happened separately on a separate line of code. However this comes at the cost of an extra commit when your merge back, an unsightly (to some) 'merge bubble', and the handling of extra branches. Additionally without pushing those branches to the remote, collaborators cannot see what you did, only the end result.

- Rebasing offers you a smooth, continuous line of work, but at the cost of obfuscating the context of work and potentially pushing rebased commits of commits that already exist in the rep, therefore consigning yourself to rebase hell (google it!). It's also more complex than merging.

Squashing Commits

PERFORCE
Version everything.

- Rebase isn't just for rebasing branches. It can be used to prune and tidy up your commits prior to pushing them to a remote repo.
- For example, you can take a series of commits, drop some, squash together others and form a beautiful and pristine codeline that will be the envy of your collaborators when you push it upstream*

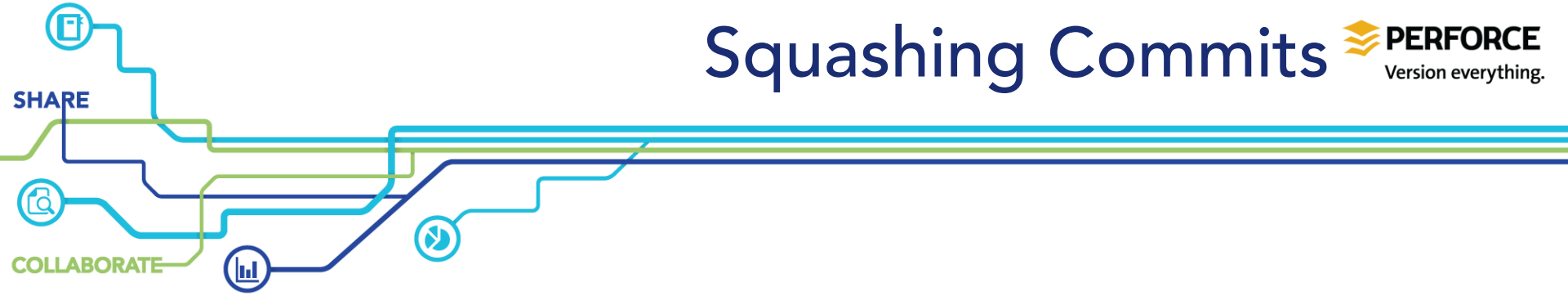
* unless you inadvertently drop or squash a commit that already exists in the remote repository. In which case you'll be vilified and scorned.

Squashing Commits

PERFORCE
Version everything.

- Let's look at an example
- We have the last commit retrieved from the remote repo at 2667 (origin/master)
- We then have four commits we made since. Most of them are just saved work, and need to be brushed under the carpet.

```
➜ mallorn git:(master) git lg
* 60b3081 - (HEAD, master)  Completed module X (2 seconds ago) <Jan Van Uytven>
* 4f93625 -  asdfasdf (33 seconds ago) <Jan Van Uytven>
* d126ee3 -  Jazz hands. FAZAAAA (66 seconds ago) <Jan Van Uytven>
* c5519d6 -  More work. (2 minutes ago) <Jan Van Uytven>
* 2667f37 - (origin/master)  Notes for forward progress (3 days ago) <Jan Van Uytven>
* e5fc471 -  Initial Commit, based on sprite-blitting (6 days ago) <Jan Van Uytven>
➜ mallorn git:(master)
```



Squashing Commits

PERFORCE
Version everything.

- To do that, call
  `git rebase --interactive HEAD~4`

- This will open an editor window with the four commits starting at the current head and going back to, but not including, HEAD~4 (which is origin/master)

Squashing Commits

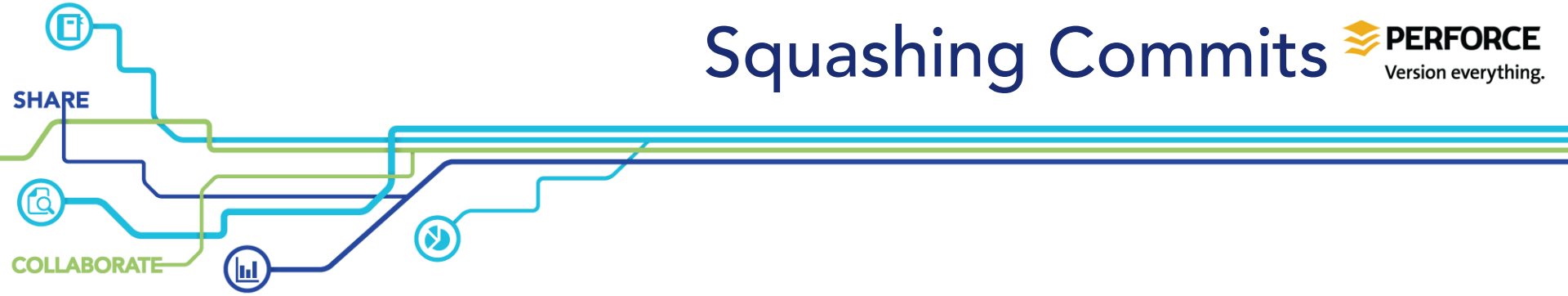
PERFORCE
Version everything.

```
1pick c5519d6 More work.
2pick d126ee3 Jazz hands. FAZAAAA
3pick 4f93625 asdfasdf
4pick 60b3081 Completed module X
5
6# Rebase 2667f37..60b3081 onto 2667f37
7#
8# Commands:
9#  c, pick = use commit
10#  r, reword = use commit, but edit the commit message
11#  e, edit = use commit, but stop for amending
12#  s, squash = use commit, but meld into previous commit
13#  f, fixup = like "squash", but discard this commit's log message
14#  x, exec = run command (the rest of the line) using shell
15#
16# These lines can be re-ordered; they are executed from top to bottom.
17#
18# If you remove a line here THAT COMMIT WILL BE LOST.
19#
20# However, if you remove everything, the rebase will be aborted.
21#
22# Note that empty commits are commented out
```

Squashing Commits

PERFORCE
Version everything.

```
pick 60b3081 Completed module X
squash c5519d6 More work.
squash d126ee3 Jazz hands. FAZAAAA
squash 4f93625 asdfasdf

# Rebase 2667f37..60b3081 onto 2667f37
#
# Commands:
#  c, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

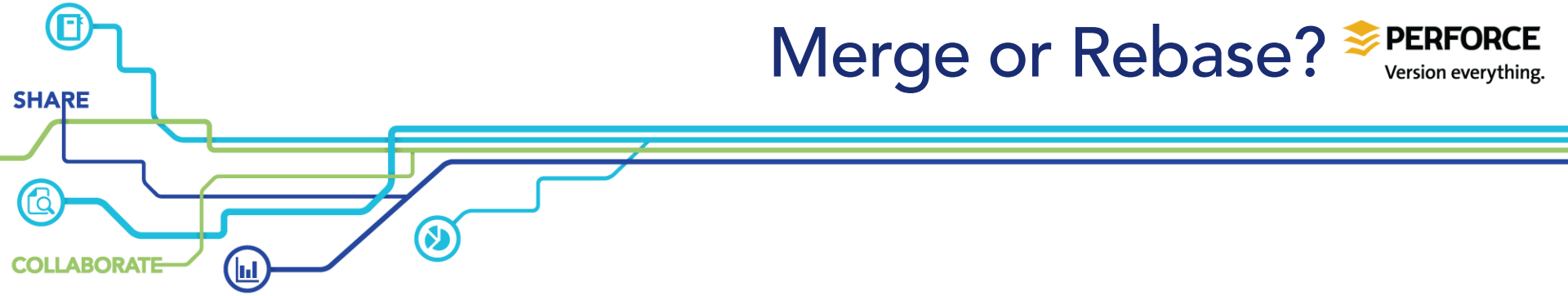


Squashing Commits

PERFORCE
Version everything.

- The commits have been folded into one, with a nice, neat commit message, ready for pushing to the remote.

```
➜ mallorn git:(master) git lg
* f8023c2 - (HEAD, master)  Completed module X (14 seconds ago) <Jan Van Uytven>
* 2667f37 - (origin/master)  Notes for forward progress (3 days ago) <Jan Van Uytven>
* e5fc471 -  Initial Commit, based on sprite-blitting (6 days ago) <Jan Van Uytven>
➜ mallorn git:(master)
```

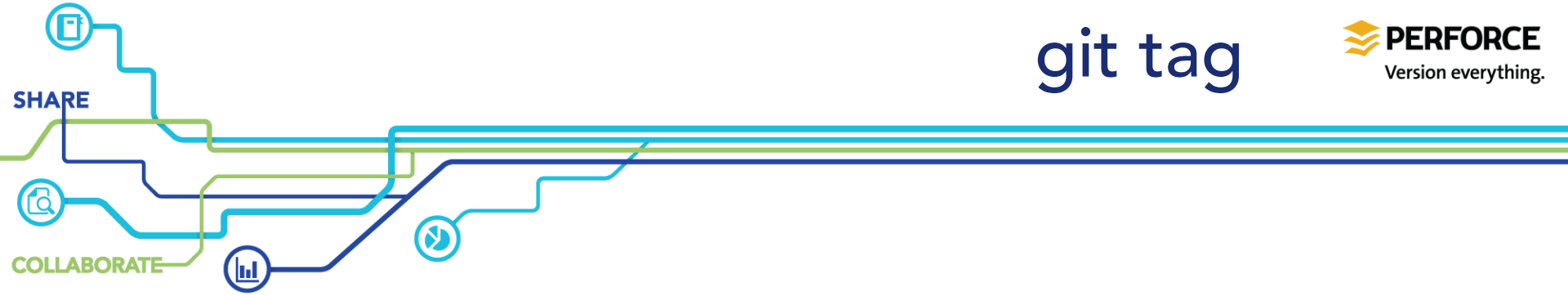


Merge or Rebase?

PERFORCE
Version everything.

- Which you use is a matter or preference or project policy.
- If you're wondering which approach Linus uses, he has stated flat out:

“I don't use rebase.”



git tag

PERFORCE
Version everything.

- Git tag allows you to create refs that point to commits or other refs.
- There are two types of tags – lightweight tags, which are just a ref, and annotated tags, which are actual objects which can contain a message (and optionally, a PGP signature)
- Lightweight tags are usually considered private to a repo

git tag

PERFORCE
Version everything.

- Annotated tags are full objects, and can be manipulated like any other blob.

```
Git tag -m “A message” tag_name commit
```

```
➜  cow-sheep git:(master) ✗ git tag -m "version 1.0" v1.0 master
➜  cow-sheep git:(master) ✗ git rev-parse v1.0
640aa6eeedf408241a7b91c30811bc2f8a93df66
➜  cow-sheep git:(master) ✗ git lg -n3
* b106933 - (HEAD, tag: v1.0, master)  Fixed typo (21 minutes ago) <Jan Van Uytven>
* 5eeef38 -  Added Farnsworth (22 minutes ago) <Jan Van Uytven>
* b7395fb -  Bender is great! (29 minutes ago) <Jan Van Uytven>%
➜  cow-sheep git:(master) ✗ git cat-file -p 640aa
object b10693309e64906e324891da1859c83c2bf478da
type commit
tag v1.0
tagger Jan Van Uytven <juytven@perforce.com> 1409157092 -0700

version 1.0
➜  cow-sheep git:(master) ✗
```

git tag

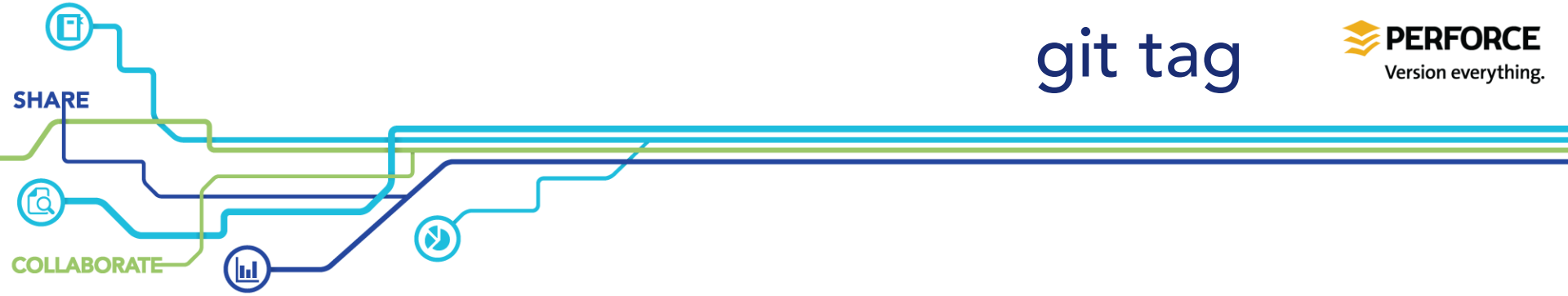
PERFORCE
Version everything.

- Lightweight tags are typically for developer use, marking commits of note
- Running rev-parse on a lightweight tag gives the commit instead of a tag object.

Git tag tag_name commit

```
➜  cow-sheep git:(master) ✗ git tag bender_is_great b7395
➜  cow-sheep git:(master) ✗ git lg -n1
* b7395fb - (HEAD, tag: temp_tag, tag: bender_is_great, master)  Bender is great! (40
minutes ago) <Jan Van Uytven>
➜  cow-sheep git:(master) ✗ git rev-parse bender_is_great
b7395fb0728ffe6cbfbba19768e970c06f5252cc
➜  cow-sheep git:(master) ✗ git cat-file -p b7395f
tree 05d3ca1b3e91f4737911387ef1d2671a4e9920d6
parent 556e52ed9ea348fcc12c36acd8f36a0c0c6a8e9a
author Jan Van Uytven <juytven@perforce.com> 1409155398 -0700
committer Jan Van Uytven <juytven@perforce.com> 1409155398 -0700

Bender is great!
➜  cow-sheep git:(master) ✗
```

•If a commit is not specified, it is assumed to be HEAD



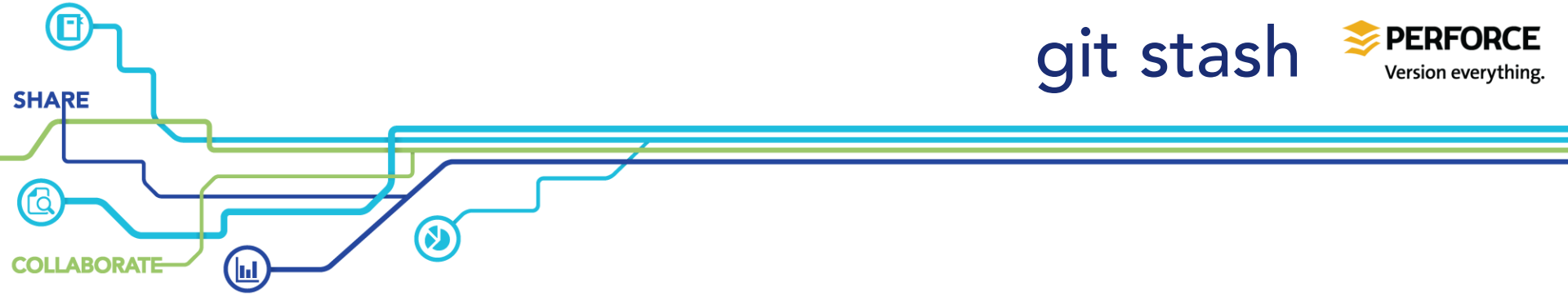
git tag

PERFORCE
Version everything.

- Delete a tag by using the -d flag

```
Git tag -d tag_name
```

```
➜  cow-sheep git:(master) ✗ git tag -d bender_is_great
Deleted tag 'bender_is_great' (was b7395fb)
```

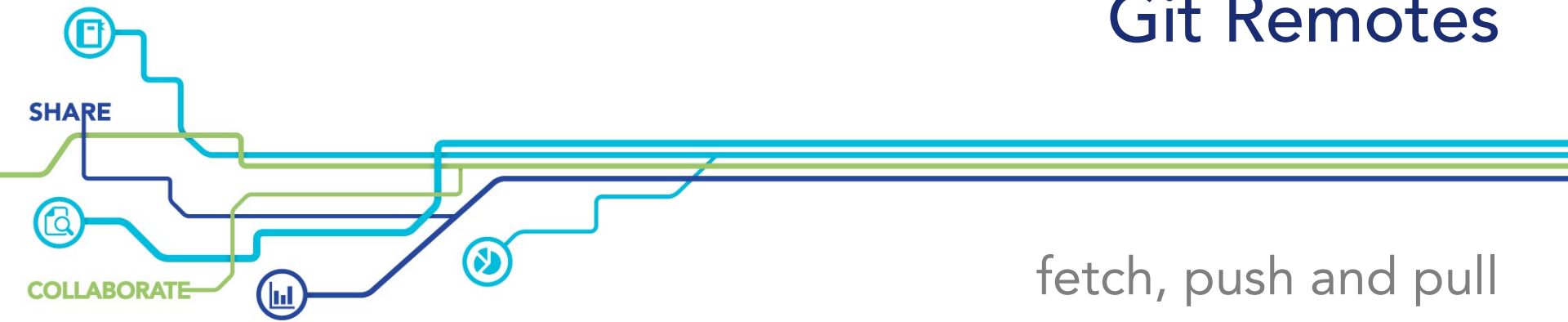


git stash

PERFORCE
Version everything.

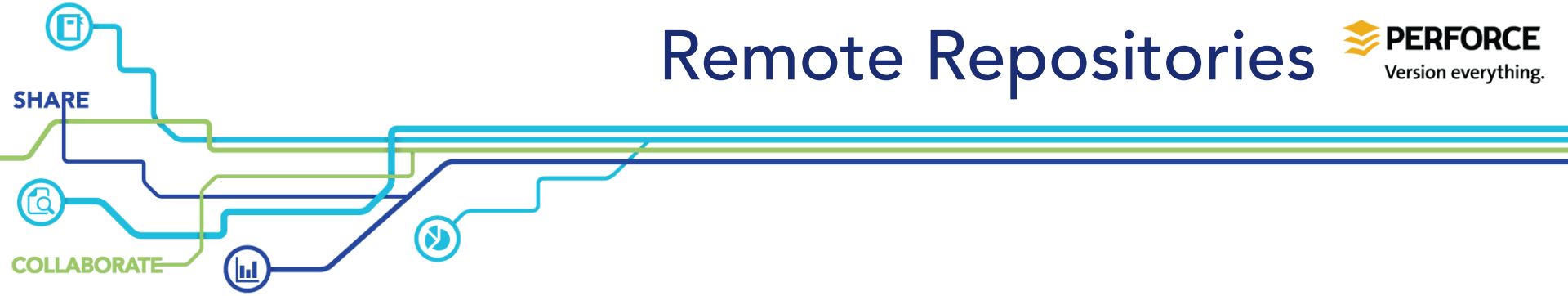
- Git stash provides a queue for hosting temporary commits that you want to save.
- You can stash more than once, with each subsequent stash going on 'top' of the previous ones.
- `git stash` save to store your work
- `git stash show` to show currently stored stashes
- `git stash pop` to remove the last stored stash and apply it to your index and working trees.

PERFORCE
Version everything.



Git Remotes

fetch, push and pull



Remote Repositories

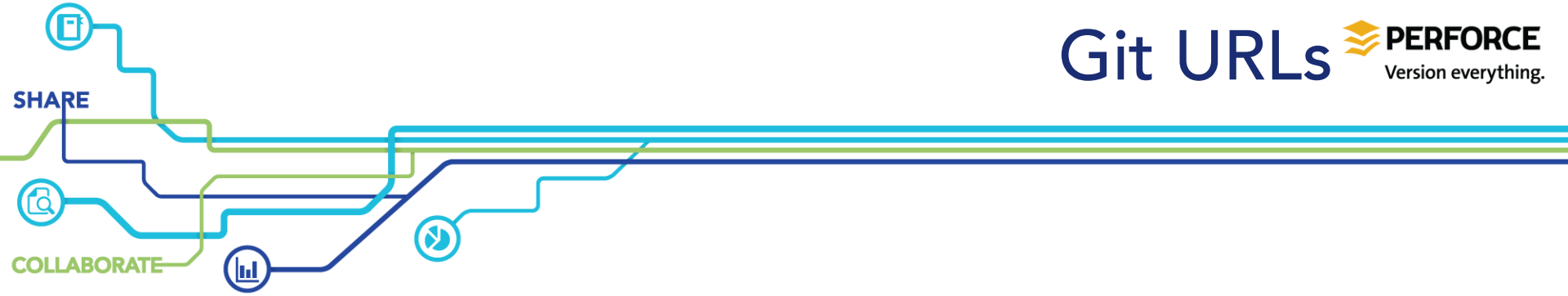
PERFORCE
Version everything.

- Git is a DVCS, and that means you can push and pull your commits (and the associated data structures) to and from other git repositories.
- Git does not impose any form of security on a repo. This is left to third-party software (SSH, HTTPS auth, file permissions).

Defining a remote repo

PERFORCE
Version everything.

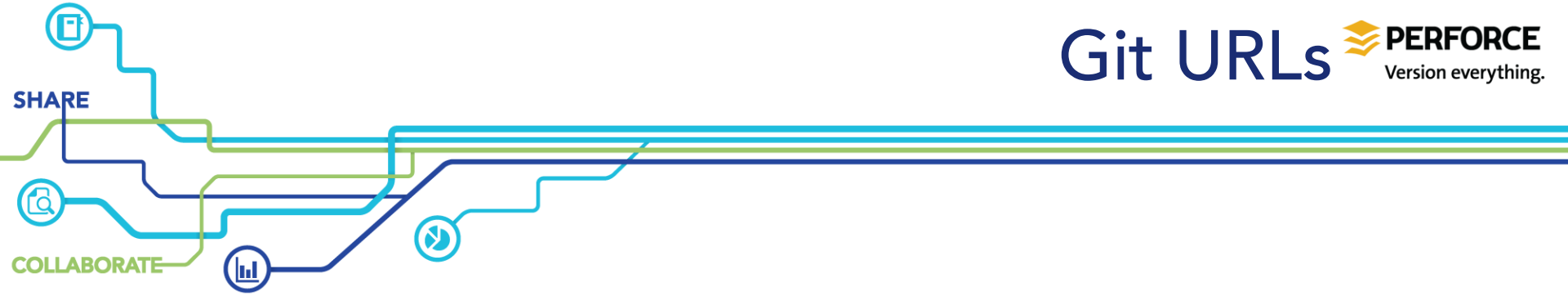
- Remote repos can be defined in one of two ways:
  - Automatically, when a `git clone` is issued
  - Manually, by using the `git remote` command.
- Use git clone when you want an intial, complete copy of the remote repo
- Use git remote when you already have a repo and want to synchronize it with the remote.



Git URLs

PERFORCE
Version everything.

- Before we can clone or define a remote in our repo, we have to be able to reference it.
- Git provides several ways of referring to a given remote, depending on how you plan to access it.



Git URLs

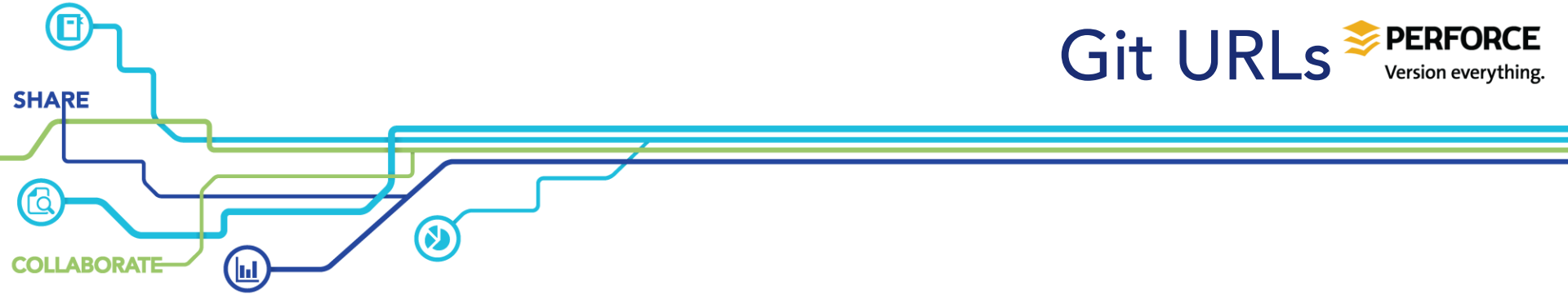
PERFORCE
Version everything.

Local filesystem: /path/to/repo.git or file:///path/to/repo.git

Git native protocol: git://example.com/path/to/repo.git

SSH: [user@]example.com:/path/to/repo.git
or ssh://[user@]example.com[:port]/path/to/repo.git

HTTP(S): http[s]://example.com/path/to/repo.git



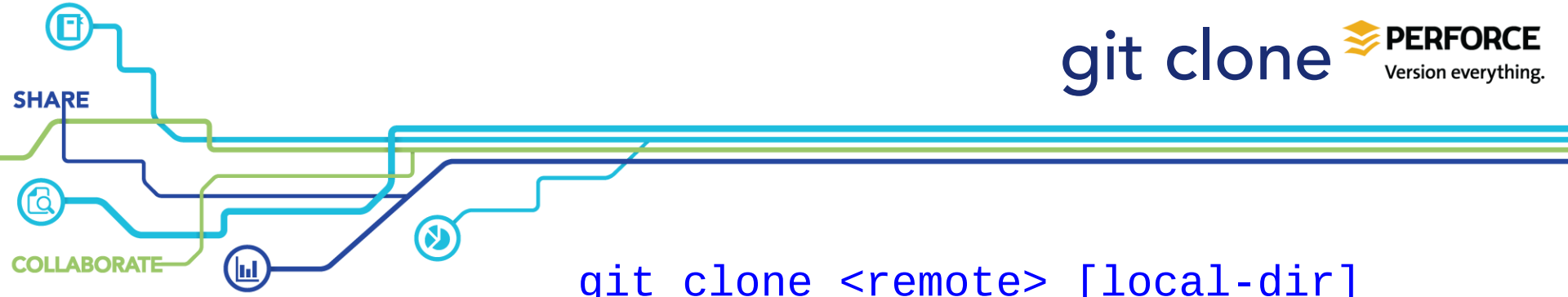
Git URLs

PERFORCE
Version everything.

- Which format to use depends on how you, or the remote provider, sets up the depot.
- Git cloud providers such as GitHub and BitBucket allow you to use either HTTPS or SSH, for example
- A local workmate may provide you with an NFS path.

git clone

- Git clone will provide you with a copy of the remote repository.
- This is not a exact copy – for example, you get copies of the remote ref's branches, but you also get the remote-tracking branches for the remote's branches.
- If the remote is bare, you get a non-bare repo by default unless you specify `--bare`

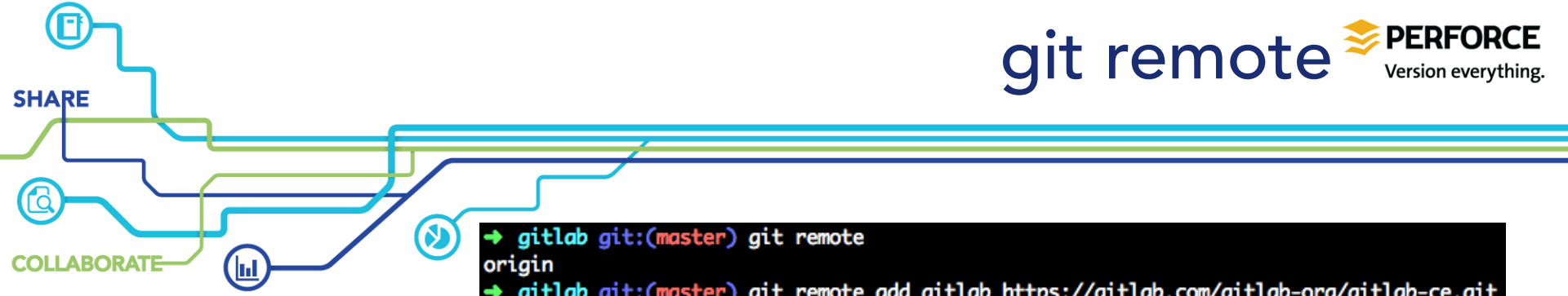


git clone

PERFORCE
Version everything.

```
git clone <remote> [local-dir]
```

```
➜ /tmp  git clone https://gitlab.com/gitlab-org/gitlab-ce.git
Cloning into 'gitlab-ce'...
remote: Counting objects: 88862, done.
remote: Compressing objects: 100% (21956/21956), done.
remote: Total 88862 (delta 67386), reused 86387 (delta 65609)
Receiving objects: 100% (88862/88862), 73.12 MiB | 620.00 KiB/s, done.
Resolving deltas: 100% (67386/67386), done.
Checking connectivity... done.
➜ /tmp
```



git remote

PERFORCE
Version everything.

```
➜  gitlab git:(master) git remote
origin
➜  gitlab git:(master) git remote add gitlab https://gitlab.com/gitlab-org/gitlab-ce.git
➜  gitlab git:(master) git remote
gitlab
origin
➜  gitlab git:(master)
```

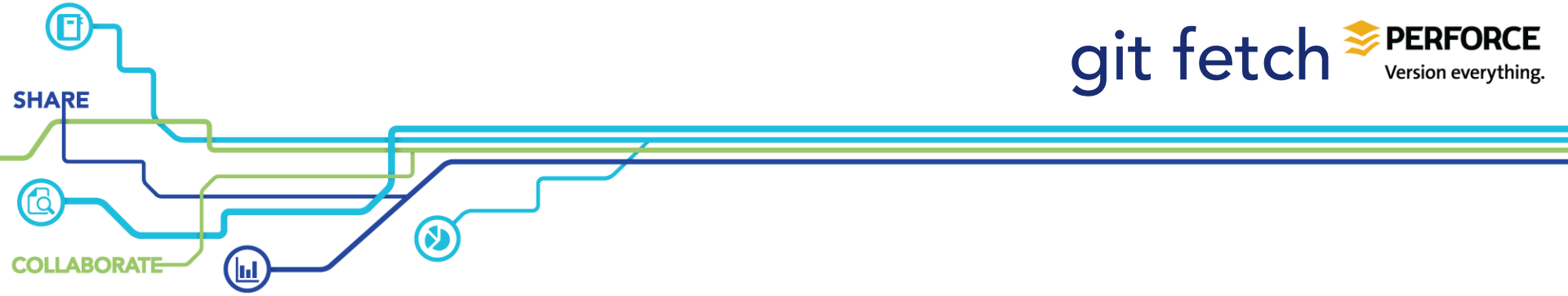
- You can also define a remote manually with `git remote add <name> <remote>`
- <remote> is the git URL for the remote repo
- <name> can be anything you want. The repo will be referred to by <name> when using git fetch and git push, and all the remote refs for it will be prefixed with it
- When cloning a repo, the remote's name is set to origin by default.

git remote

PERFORCE
Version everything.

```
➜ gitlab git:(master) git remote show gitlab
* remote gitlab
  Fetch URL: https://gitlab.com/gitlab-org/gitlab-ce.git
  Push  URL: https://gitlab.com/gitlab-org/gitlab-ce.git
  HEAD branch: master
  Remote branches:
    5-4-stable  new (next fetch will store in remotes/gitlab)
    6-0-stable  new (next fetch will store in remotes/gitlab)
    6-1-stable  new (next fetch will store in remotes/gitlab)
    6-2-stable  new (next fetch will store in remotes/gitlab)
    6-3-stable  new (next fetch will store in remotes/gitlab)
    6-4-stable  new (next fetch will store in remotes/gitlab)
    6-5-stable  new (next fetch will store in remotes/gitlab)
    6-6-stable  new (next fetch will store in remotes/gitlab)
    6-7-stable  new (next fetch will store in remotes/gitlab)
    6-8-stable  new (next fetch will store in remotes/gitlab)
    6-9-stable  new (next fetch will store in remotes/gitlab)
    7-0-stable  new (next fetch will store in remotes/gitlab)
    7-1-stable  new (next fetch will store in remotes/gitlab)
    7-2-stable  new (next fetch will store in remotes/gitlab)
    gitlab-flow new (next fetch will store in remotes/gitlab)
    master      new (next fetch will store in remotes/gitlab)
  Local ref configured for 'git push':
    master pushes to master (local out of date)
➜ gitlab git:(master)
```

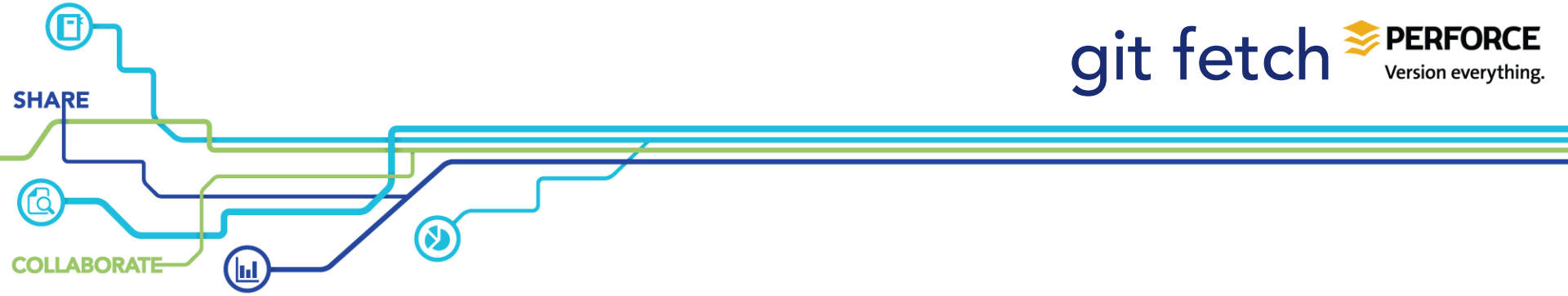
- Using git remote with no options will give you the list of remotes associated with your repo
- You can use `git remote show <name>` to get more information about that remote
- Remotes are one-way only – Remote repos are not aware of who is tracking them.
- Remote repos never receive any information from other repos that aren't listed as remotes themselves for that repo.
- Remove remotes with `git remote remove <name>`



git fetch

PERFORCE
Version everything.

- Once we have a remote defined, we can start fetching and pushing changes to and from that remote repository.

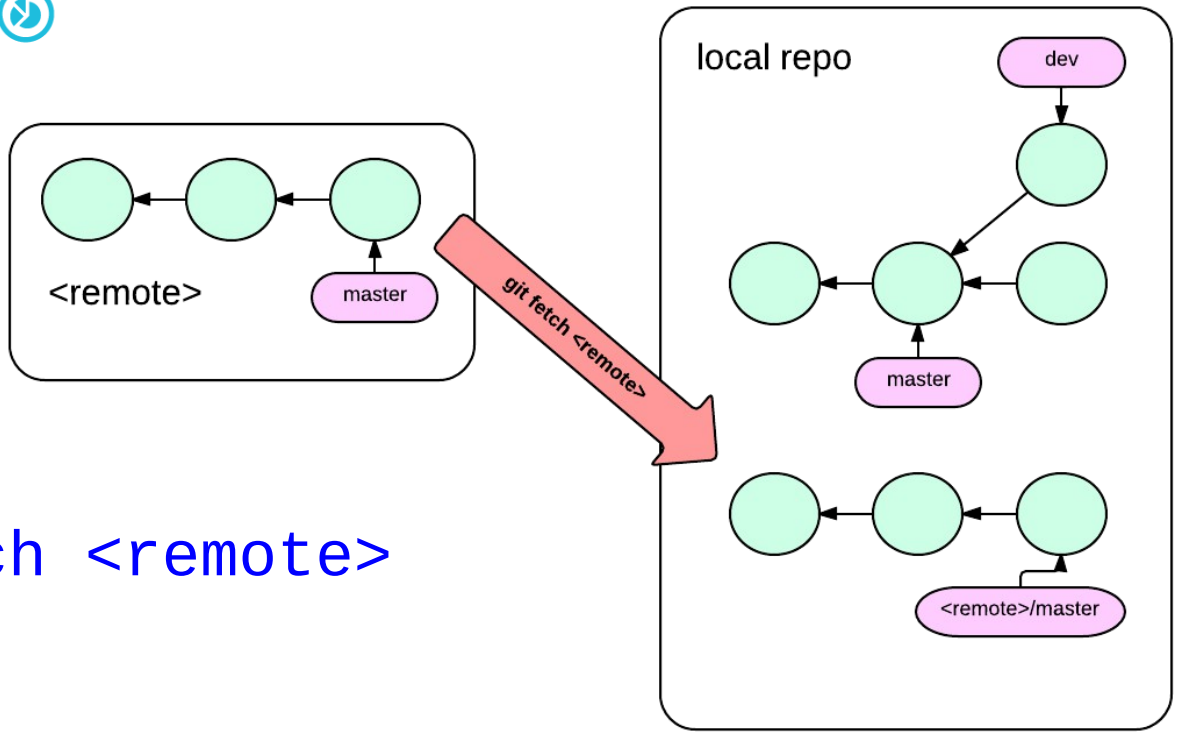


git fetch

PERFORCE
Version everything.

- Git fetch is used to obtain the objects and refs from another repository
- If the remote has not been fetched from before, then the entire contents of the repo will be downloaded. Otherwise only any changed/new refs and new commits since the last fetch are obtained.
- New refs are created to point to the branch heads of the remote repo's commits. These refs are known as remote refs, or remote-tracking branches
- Fetching does not fold the remote repo's changes into your existing repo tree.

git fetch



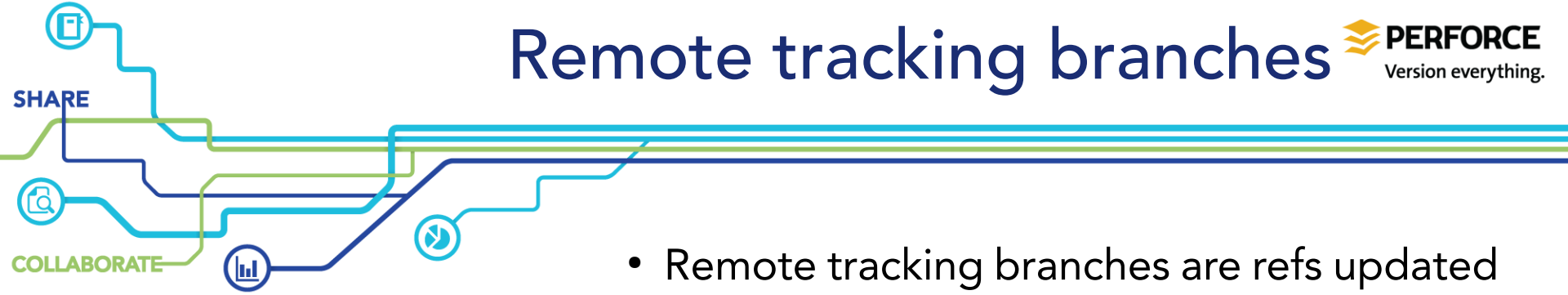

```
git fetch <remote>
```

git fetch

PERFORCE
Version everything.

```
➜ gitlab git:(master) git fetch --all
Fetching origin
remote: Counting objects: 2015, done.
remote: Compressing objects: 100% (1664/1664), done.
remote: Total 2015 (delta 1503), reused 398 (delta 321)
Receiving objects: 100% (2015/2015), 325.81 KiB | 340.00 KiB/s, done.
Resolving deltas: 100% (1503/1503), completed with 285 local objects.
From https://gitlab.com/gitlab-org/gitlab-ce
   c0e1312..e38081e  master     -> origin/master
 * [new branch]      7-2-stable -> origin/7-2-stable
 * [new branch]      gitlab-flow -> origin/gitlab-flow
 * [new tag]         v7.2.0     -> v7.2.0
 * [new tag]         v7.2.0.rc1 -> v7.2.0.rc1
 * [new tag]         v7.2.0.rc2 -> v7.2.0.rc2
 * [new tag]         v7.2.0.rc3 -> v7.2.0.rc3
 * [new tag]         v7.2.0.rc4 -> v7.2.0.rc4
 * [new tag]         v7.2.0.rc5 -> v7.2.0.rc5
➜ gitlab git:(master)
```

- Example: fetching all recent commits made to the GitLab community repo
- `--all` is used to fetch from all remotes, if more than one is specified
- Commits downloaded are listed as a range – In this case, commits from e38081e back to, but not including, c0e1312.



Remote tracking branches

PERFORCE
Version everything.

```
➜  gitlab git:(master) tree .git/refs/remotes
.git/refs/remotes
└── origin
    ├── 7-2-stable
    ├── HEAD
    ├── gitlab-flow
    └── master

1 directory, 4 files
➜  gitlab git:(master)
```

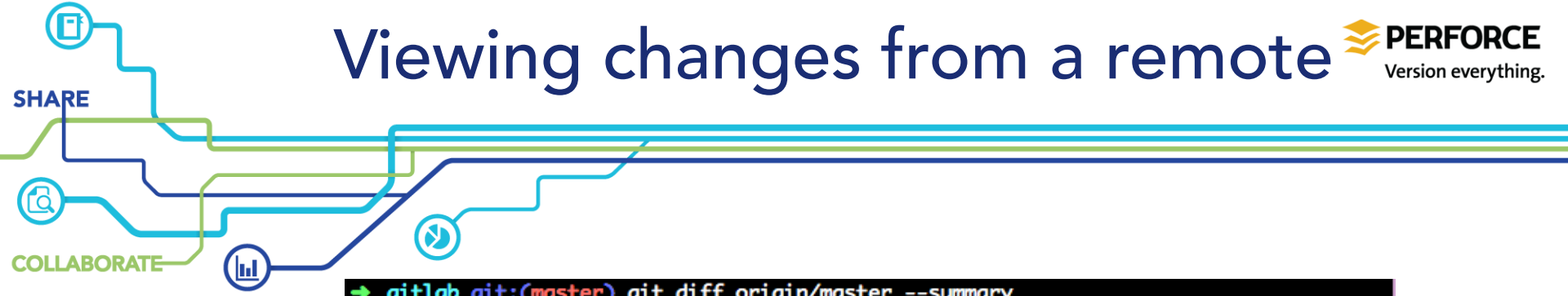
- Remote tracking branches are refs updated with the contents of the remote's local branches.
- The are stored in .git/refs/remotes, under the remote name
- For example, the remote master branch's ref would be stored locally as .git/refs/remotes/origin/master
- They are typically referenced as origin/master, origin/HEAD, etc...

Viewing changes from a remote

PERFORCE
Version everything.

- Once you have fetched the new commits from the remote repo, you can examine the changes made since the last fetch/pull.
- The remote HEAD is stored in FETCH_HEAD
- git log master..origin/master will provide you with the list of commits in origin/master that you don't have in master.
- You can also use git diff origin/master --summary to view the differences between origin/master and your local HEAD. --summary just condenses the effect of the diff into one line (create, delete, etc..)

Viewing changes from a remote



PERFORCE
Version everything.

```
➜  gitlab git:(master) git diff origin/master --summary
 delete mode 100644 app/assets/javascripts/behaviors/toggle_diff_line_wrap_behavior.co
ffee
 delete mode 100644 app/assets/javascripts/diff.js.coffee
 delete mode 100644 app/assets/javascripts/labels.js.coffee
 delete mode 100644 app/assets/stylesheets/generic/timeline.scss
 delete mode 100644 app/services/archive_repository_service.rb
 delete mode 100644 app/views/projects/blob/diff.html.haml
 delete mode 100644 app/views/projects/commits/diffs/_match_line.html.haml
 delete mode 100644 app/views/search/results/_empty.html.haml
 create mode 100644 db/fixtures/development/08_wall.rb
 delete mode 100644 doc/api/labels.md
 create mode 100644 doc/update/6.0-to-7.1.md
 delete mode 100644 doc/update/6.0-to-7.2.md
 delete mode 100644 doc/update/7.1-to-7.2.md
 delete mode 100644 doc/workflow/labels.md
 delete mode 100644 doc/workflow/labels/label1.png
 delete mode 100644 doc/workflow/labels/label2.png
 delete mode 100644 doc/workflow/labels/label3.png
 create mode 100644 features/dashboard/search.feature
 delete mode 100644 features/search.feature
 create mode 100644 features/steps/dashboard/search.rb
```

Merging changes from a remote

PERFORCE
Version everything.

- Once you have the remote commits, you can merge them into your local repo tree.
- You can do this using a merge or a rebase.
- Either way, the process is identical to a normal merge or rebase – the only difference is that you are using a remote tracking branch as one of the sources.

Merging changes from a remote

PERFORCE
Version everything.

- Sometimes, a fast-forward merge is all you need to do

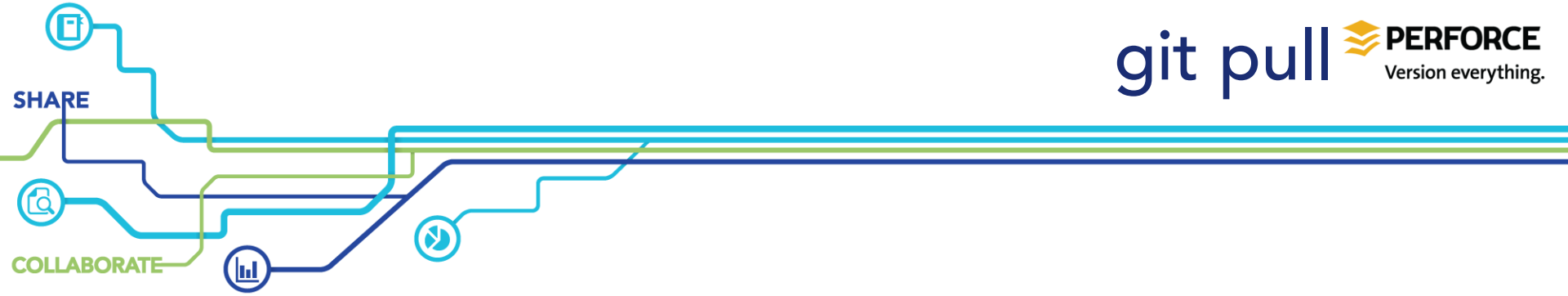
```
➜ gitlab git:(master) git merge --ff-only origin/master
Updating c0e1312..e38081e
Fast-forward
 .travis.yml                                        |   7 +-
 CHANGELOG                                          |  33 ++-
 GITLAB_SHELL_VERSION                               |   2 +-
 Gemfile                                            |   3 +-
 Gemfile.lock                                       |  33 +--
 PROCESS.md                                         |   4 +-
 VERSION                                            |   2 +-
 app/assets/javascripts/application.js.coffee       |  34 ++-
 .../behaviors/toggle_diff_line_wrap_behavior.coffee |  14 ++
 app/assets/javascripts/diff.js.coffee              |  47 ++++
 app/assets/javascripts/dispatcher.js.coffee        |  12 +-
 app/assets/javascripts/labels.js.coffee            |  35 +++
 app/assets/javascripts/notes.js.coffee             |   4 +-
 app/assets/javascripts/pager.js.coffee             |  22 +-
 app/assets/stylesheets/generic/common.scss         |  17 +-
 app/assets/stylesheets/generic/timeline.scss       |  77 ++++++
 app/assets/stylesheets/sections/commits.scss       |   1 +
 app/assets/stylesheets/sections/dashboard.scss     |   4 +
```

Merging changes from a remote

PERFORCE
Version everything.

- Sometimes you will want to, or be forced to, create a merge commit.
- But in the end, it's no different from a merge or rebase from a local branch.

```
➜ gitlab git:(master) git merge --no-ff origin/master
Removing spec/support/valid_commit_with_alt_email.rb
Removing spec/support/valid_commit.rb
Removing spec/support/big_commits.rb
Auto-merging spec/services/projects/fork_service_spec.rb
Removing spec/seed_project.tar.gz
Removing spec/helpers/tree_helper_spec.rb
Removing features/steps/dashboard/search.rb
Removing features/dashboard/search.feature
Auto-merging doc/update/6.0-to-7.2.md
Removing db/fixtures/development/08_wall.rb
Merge made by the 'recursive' strategy.
 .travis.yml                                        |   7 +-
 CHANGELOG                                          |  33 ++-
 GITLAB_SHELL_VERSION                               |   2 +-
 Gemfile                                            |   3 +-
 Gemfile.lock                                       |  33 +--
 PROCESS.md                                         |   4 +-
 VERSION                                            |   2 +-
 app/assets/javascripts/application.js.coffee       |  34 ++-
 .../behaviors/toggle_diff_line_wrap_behavior.coffee |  14 ++
 app/assets/javascripts/diff.js.coffee              |  47 ++++
 app/assets/javascripts/dispatcher.js.coffee        |  12 +-
 app/assets/javascripts/labels.js.coffee            |  35 +++
 app/assets/javascripts/notes.js.coffee             |   4 +-
 app/assets/javascripts/pager.js.coffee             |  22 +-
 app/assets/stylesheets/generic/common.scss         |  17 +-
 app/assets/stylesheets/generic/timeline.scss       |  77 ++++++
 app/assets/stylesheets/sections/commits.scss       |   1 +
```



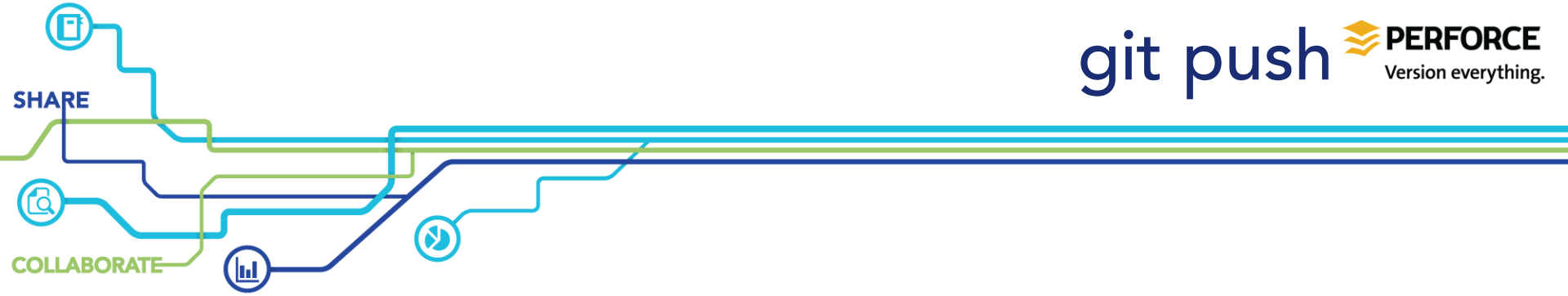
git pull

PERFORCE
Version everything.

- `git pull` combines two commands into one:
  ```
  git fetch
  git merge FETCH_HEAD
  ```
- Git pull will fetch the remote commits and automatically attempts to merge them into your current branch.
- So if you are on master, and you issue a git pull, you fetch the commits from the remote as normal, and then automatically attempt to git merge origin/master

git pull

- Be careful using git pull. It's not intrinsically dangerous, but merging without viewing what you're merging first can be undesirable.



git push

PERFORCE
Version everything.

- `git push` is the opposite of `git fetch`.
- `git push <remote> <branch>` will upload all the commits and refs for <branch> to the remote repo
- `git push --all` pushes everything upstream, to all remotes
- Specify `-n` to do a dry run, to check for potential problems.
- If you don't specify a remote, origin is used by default.

git push

PERFORCE
Version everything.

- Notice here that despite claiming to be one conflict behind, we still ran into a conflict.
- This is because git doesn't know what's on the remote repo until it tries to contact it.
- The solution, as git helpfully suggests, is to fetch the latest commits and merge them into your repo.

```
➜  sprite-blitting git:(master) git status -sb
## master...origin/master [ahead 1]
➜  sprite-blitting git:(master) git push --all
To git@github.com:ysgard/sprite-blitting.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:ysgard/sprite-blitting.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

git push

PERFORCE
Version everything.

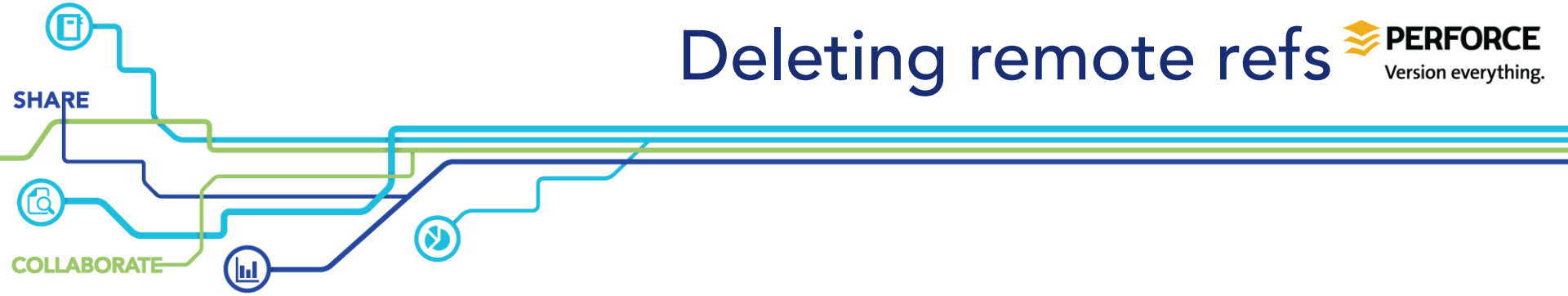
```
➜ sprite-blitting git:(master) git st
## master...origin/master [ahead 2]
➜ sprite-blitting git:(master) git push --all
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 573 bytes | 0 bytes/s, done.
Total 5 (delta 3), reused 0 (delta 0)
To git@github.com:ysgard/sprite-blitting.git
   992caba..1033cdc  master -> master
➜ sprite-blitting git:(master)
```

```
➜ sprite-blitting git:(master) git fetch --all
Fetching origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From github.com:ysgard/sprite-blitting
   0c215bc..992caba  master     -> origin/master
➜ sprite-blitting git:(master) git st
## master...origin/master [ahead 1, behind 1]
➜ sprite-blitting git:(master) git diff origin/master --summary
➜ sprite-blitting git:(master) git merge origin/master
Merge made by the 'recursive' strategy.
 source/app.d | 13 +++++++++++++
 1 file changed, 13 insertions(+)
```

git push

PERFORCE
Version everything.

- Always resolve conflicts with the remote by fetching/merging/rebasing the remote's commits.
- Although git allows you to force push (-f), there are few, if any, cases where this is justified.



Deleting remote refs

PERFORCE
Version everything.

- You can delete a remote tag or branch by using
  `git push <remote> :<branch|tag>`
- For example the command
  `git push origin :develop`
  will delete the branch develop on the remote repo
- Needless to say, this can be disruptive to other collaborators, so exercise caution.

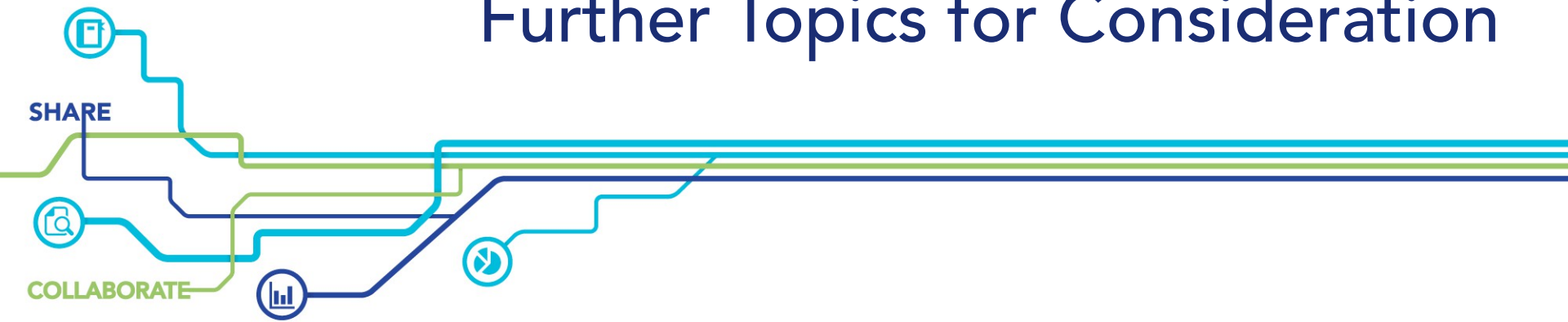
git Etiquette

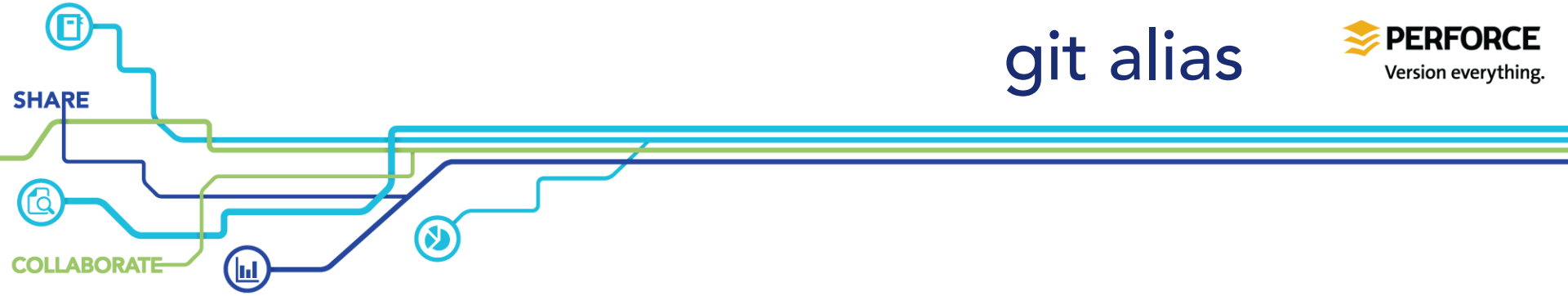
PERFORCE
Version everything.

- The reason why it's preferable to push to a bare repo is because developer disruption can be minimized
- In general, never push to another developer's repo. Always request that your changes be fetched/merged.
- This is known as a *pull request*, and many git management tools (and cloud services) offer facilities to quickly request and resolve pull requests.

PERFORCE
Version everything.

Further Topics for Consideration





git alias

PERFORCE
Version everything.

- Use git alias to provide shortcuts for your most common commands:
  `git alias –global alias.<shortcut> “command”`

```
alias.co=checkout
alias.ds=diff --staged
alias.st=status -sb
alias.amend=commit --amend
alias.undo=reset --soft HEAD^
```

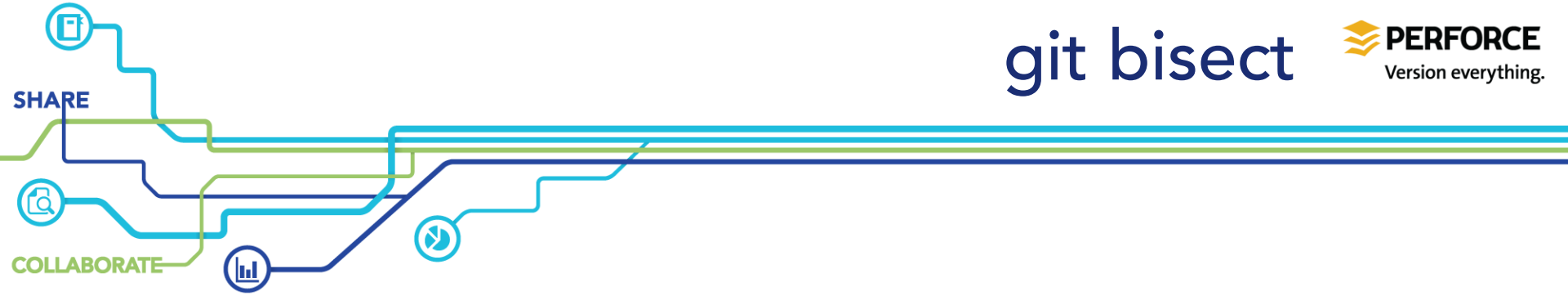
git cherry-pick, git revert

PERFORCE
Version everything.

- `git cherry-pick <commit>` – allows you to pull <commit>, which can be from anywhere, into your current branch as a copy of that commit.
- `git revert <commit>` – Applies the inverse of <commit>, in essence creating a new commit on the current branch that completely undoes the effects of <commit>

git blame

- Nobody is perfect.
- Bugs happen, and sometimes you need to find out who to blame.
- `git blame -L<begin>,<end> <file>` will show who was responsible for writing the lines from <begin> to <end> in that file.

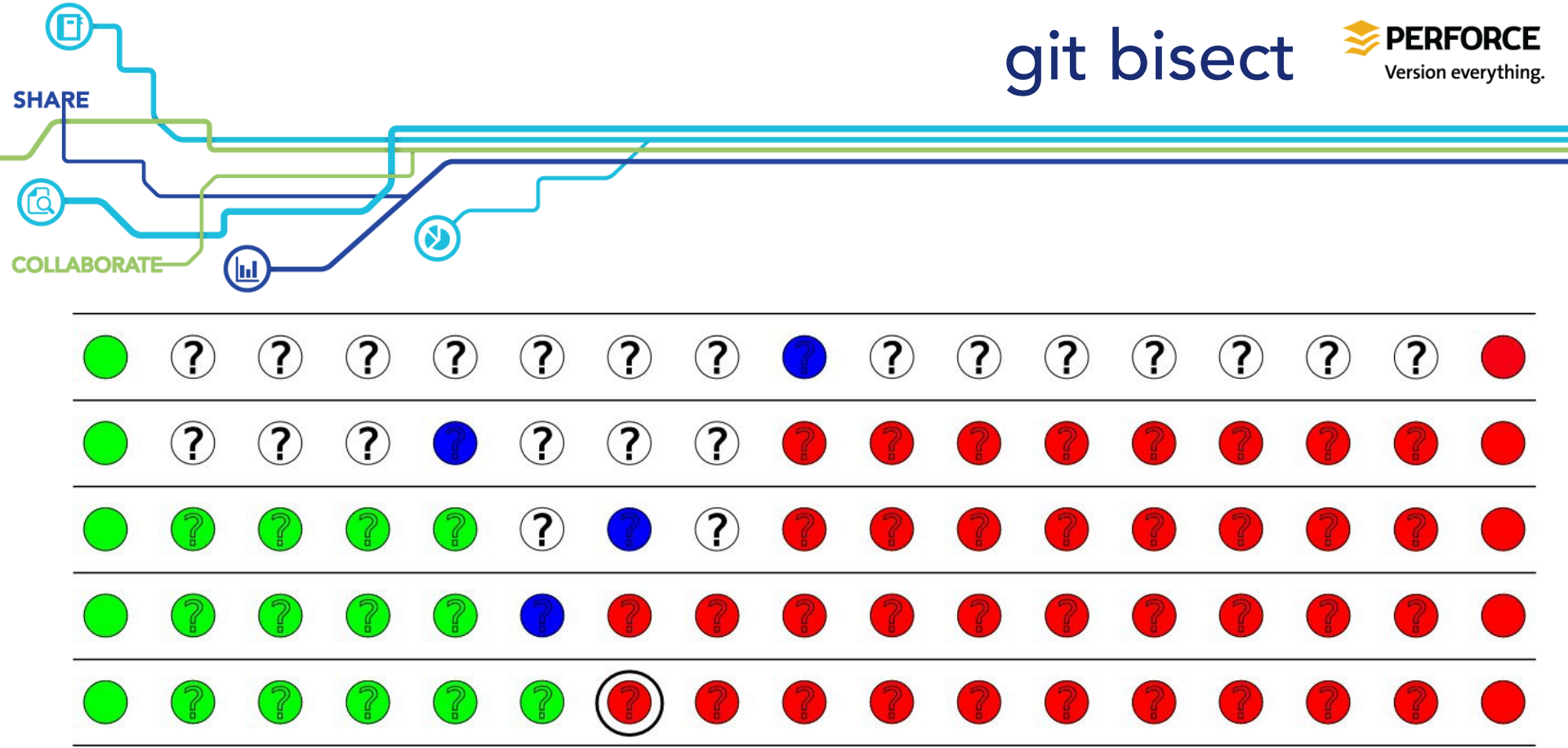


git bisect

PERFORCE
Version everything.

- Git bisect is an interactive command that uses a binary search to search from a 'bad' commit to a known good one
- With each iteration, you test the provided commit and tell git bisect whether it is good or bad.
- Git bisect will stop when it can specify the exact commit that caused the problem.

git bisect

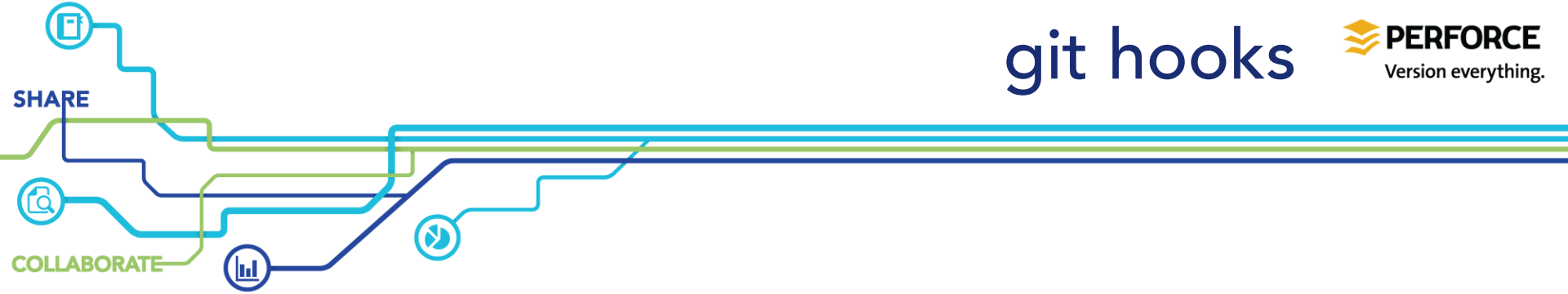


git reflog

PERFORCE
Version everything.

```
→ mallorn git:(master) git reflog
f8023c2 HEAD@{0}: rebase -i (finish): returning to refs/heads/master
f8023c2 HEAD@{1}: rebase -i (continue): Completed module X
2667f37 HEAD@{2}: rebase -i (start): checkout HEAD~4
60b3081 HEAD@{3}: rebase: aborting
2667f37 HEAD@{4}: rebase -i (start): checkout HEAD~4
60b3081 HEAD@{5}: rebase: aborting
2667f37 HEAD@{6}: rebase -i (start): checkout HEAD~4
60b3081 HEAD@{7}: rebase: aborting
60b3081 HEAD@{8}: rebase: aborting
60b3081 HEAD@{9}: commit: Completed module X
4f93625 HEAD@{10}: commit: asdfasdf
d126ee3 HEAD@{11}: commit: Jazz hands. FAZAAAA
c5519d6 HEAD@{12}: commit: More work.
2667f37 HEAD@{13}: commit: Notes for forward progress
e5fc471 HEAD@{14}: commit (amend): Initial Commit, based on sprite-blitting
38b4aa7 HEAD@{15}: commit (amend): Initial Commit
753d124 HEAD@{16}: commit (initial): Initial Commit
→ mallorn git:(master)
```

- If you ever lose any commits, or accidentally delete a branch, or screw up a rebase, you can get a full record of all commits involved by taking advantage of Git's reflog, which is a running journal of all manipulations made in git.
- `git reflog`



git hooks

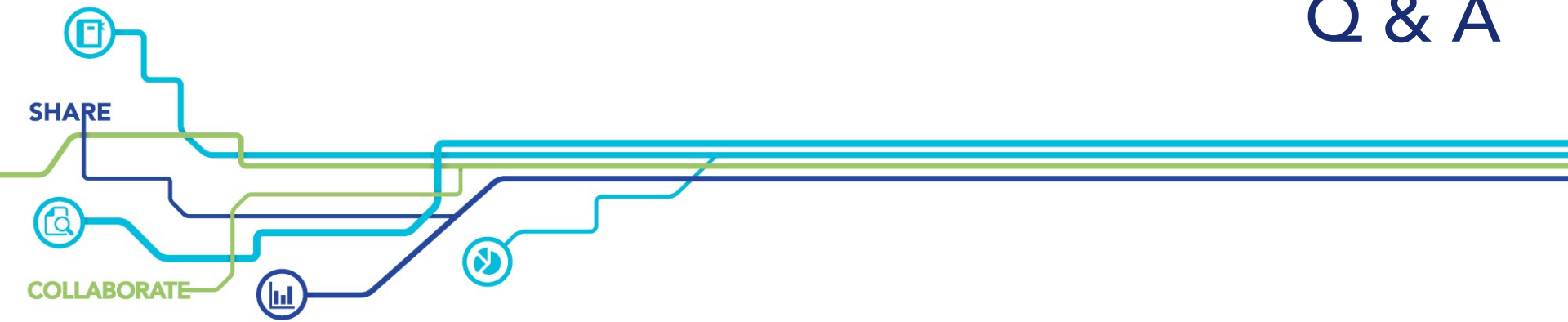
PERFORCE
Version everything.

```
➜  git_presentation git:(master) ✗ tree .git/hooks
.git/hooks
├── applypatch-msg.sample
├── commit-msg.sample
├── post-update.sample
├── pre-applypatch.sample
├── pre-commit.sample
├── pre-push.sample
├── pre-rebase.sample
├── prepare-commit-msg.sample
└── update.sample
```

- Git hooks are scripts that can be called when various actions are taken
- There are client-side git hooks (for commits, merges, etc...) as well as server-side hooks (for pushes).
- All are stored in .git/hooks/<hook_action>
- `git help hooks`

PERFORCE

Version everything.



Q & A