Revision Selection - Ancestry References

Suppose we have the following commits:

```
$ git log --pretty=format:'%h %s' --graph
*   2e25043 Merge pull request #18 from ...
|\
| * 4950521 fix sim by normalizing SNP
columns
| * 69402cd generate g effects
|/
* c455717 tabulate_output.py
* f3fe695 Merge pull request #17 from ...
```

A caret **^** at the end of a reference refers to the parent of that commit. e.g. `c455717^` will refer to `f3fe695`

For merge commits such as `2e25043`, `2e25043^` will refer to its first parent, `c455717` while `2e25043^2` will refer to its second parent `4950521`.

Moreover, since HEAD points to `2e25043`, `HEAD^` and `2e25043^` are equivalent.

A tilde ~ also refers to the first parent, so HEAD^ and HEAD~ are equivalent.

HEAD~2 refers to the first parent of the first parent of HEAD, and the same pattern goes for HEAD~3 etc.

Revision Selection - Commit Ranges

`git log master..experiment`

will show commits in experiment not reachable from master so the output will be

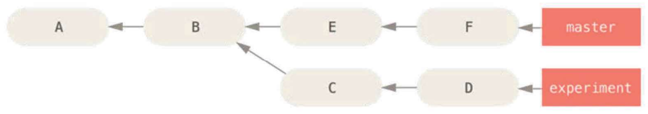
```
D
C
```

On the other hand,

`git log experiment..master`

will output

```
F
E
```



Rewriting History

In order to rewrite the last few commits, use the git interactive rebase, e.g.

`git rebase -i HEAD~3`

will rebase the last 3 commits onto the 4th oldest commit.

You will see an editor open up with texts like the ones in the next slide. Beware the the commits are listed in the reverse order to that of git log.

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

Three Trees

HEAD
Last commit snapshot, next parent. HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch.

Index
Proposed next commit snapshot. The index is your proposed next commit. We've also been referring to this concept as Git's "Staging Area" as this is what Git looks at when you run git commit.

Working Directory
The actual directories with files that you can modify with an editor.

Git Plumbing Commands

Plumbing commands refer to a set of git commands that do low-level work and are not usually used for ordinary purposes.

For example, **git hash-object** is a plumbing command:

```
echo 'test content' | git hash-object -w --stdin
```

--stdin indicates taking input from stdin, and -w indicates writing the content into a new file in .git/objects

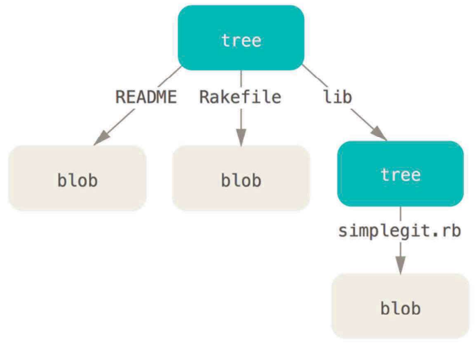
Git Objects

```
echo 'version 1' > test.txt
git hash-object -w test.txt
# returns a hash, say 83baae61804e65cc73a7201a7252750c76066a30

echo 'version 2' > test.txt
git hash-object -w test.txt
# returns another hash, say 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

find .git/objects -type f
# .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
# .git/objects/83/baae61804e65cc73a7201a7252750c76066a30
# ...

git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30
# prints version 1
git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
# prints version 2

git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
# prints "blob", the type of the file, due to the -t option
# blob is the leaf of the directory tree, representing a regular file.
```

Tree Objects

Another type of git objects is a tree object, which represents directory entries.

For example, each commit will point to a tree object, which is the top directory of the repository when the commit snapshot is taken.



Commit Objects

Each commit is also stored as a git object, which will contain information such as the tree object is points to, the parent commit(s), author, commit message, etc.

```
$ git cat-file -p 02250b1e77c88a20fba80bad2a76bb737d2d59e9
                  tree 671ea1871bdeb5f27881428c00abb4cc12a3238c
                  parent 55bb5940a549a475c768b503cb1b76a6f029d444
                  author Aaron <rustinante@gmail.com> 1574341950 -0800
                  committer Aaron <rustinante@gmail.com> 1574341950 -0800
                  b1 4
```

Git References

Inside .git/refs/ there are files whose names are aliases to commit hashes.

For example, .git/refs/heads/master will contain the commit hash of the commit pointed to by the master branch.

Another set of references are the remotes.

For example, refs/remotes/origin/master will contain the commit hash pointed to by the master branch in the remote named origin.

Git Refspec

When we do something like `git remote add origin some-url`, an entry is created in the **.git/config** file:

```
[remote "origin"]
        url = some-url
        fetch = +refs/heads/*:refs/remotes/origin/*
```

The fetch has the format +<source>:<destination>, where <source> is the references on the remote, and <destination> is where those references will be tracked locally.
The + sign means update the local references even if the remote branch update is not a fast forward, e.g. if originally we have commits **A <-- B (origin/master)(master)**, then somebody viciously changed history on the remote to A <-- C, the + sign says this is okay and update the local references to be

Topological Order

A **topological order** on a directed acyclic graph (DAG) G = (V, E) is a **total order** on all of its vertices such that if there is a directed edge from **v1** to **v2**, then **v1** < **v2**.

We can view a topological ordering/sorting on G as arranging the vertices on a horizontal line such that all edges go from left to right.

Depth-First Search (DFS) for Both Directed and Undirected Graphs

```
dfs(G = (V, E), s):
    for each vertex u in V:
        u.color = white
        u.discoverer = None
    time = 0
    for each vertex u in G.adj[s]:
        if u.color == white:
            dfs_visit(u)
```

```
dfs_visit(u):
    time = time + 1
    u.discover_time = time
    u.color = gray
    for each v in G.adj[u]:
        if v.color == white:
            v.discoverer = u
            dfs_visit(v)
    u.color = black
    time = time + 1
    u.finish_time = time
```

This is the pseudo code.

- white means undiscovered
- gray means being processed
- black means finished processing

G.adj[**u**] is the set of vertices reachable from **u**.

A stack can (and probably should) be used to implement DFS instead of recursion.

In fact, a stack should be used in Python to implement DFS due to the interpreter's limit on recursion depth.

Python DFS Implementation Using a Stack

Depending on your need, each element on the stack can be a vertex coupled with some extra info.

For example, if you want to keep track of the path, something like this will help:

```
stack = [(vertex, [vertex])]
visited = set()
while stack:
    v, path = stack.pop()
    visited.add(v)
    do something...
    for child in v.adjacent_vertices:
        if child not in visited:
            stack.append((child, path + [child]))
        do something...
```