Essay: The Inner Workings of Git

Hugo Heyman - hheyman@kth.se
Theodor Moise - moise@kth.se

April 22, 2021

1 Introduction

At the time of writing Git is the hegemonic version control system. It is used in software development all over the world, and has been for years now. We aim to efficiently bring insight of Git's inner workings by describing its data-, object- and branching model; followed by a discussion of the advantages and disadvantages of its design. We conclude that Gits distributed design enabled by its atomic object model enjoys success for good reason, yet its design is arguably not without fault.

2 The structure of Git

Linus Torvalds [2] designed git to maximise the benefits of using version control. Git had to be fast, for example when branching and merging. Therefore git's design needed to be different from previous VCS. The result is the git object model. At its core, the structure of git is simply that of a directed acyclic graph (DAG), illustrated below. The individual nodes consist of 'snapshots' [14], more commonly known as *commits*. The DAG design of the object model arises from git's functionality regarding branching and merging these snapshots.

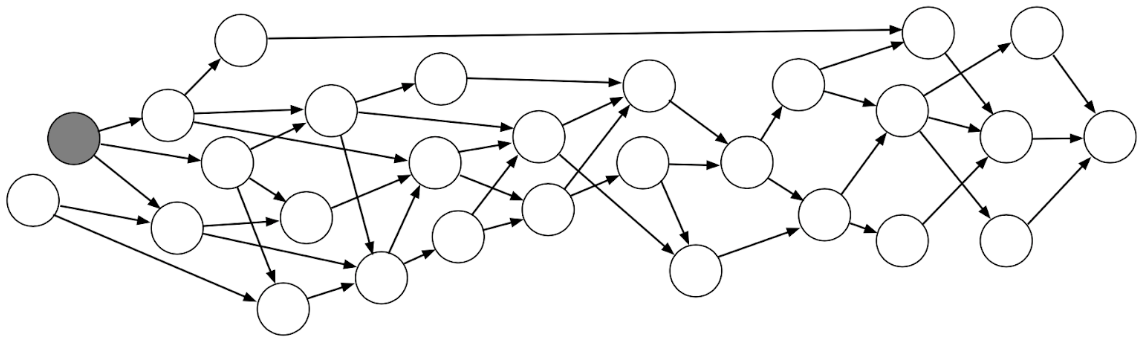

Figure 1: An example of a directed acyclic graph (DAG)

2.1 Git's data model

Using Git's terminology, a file in a directory is called a *blob* and treated as a binary file. A directory is the *tree* which maps names (strings) to either blobs or other trees, enabling trees to accommodate other trees. Linus Torvalds designed the directory structure much like a standard UNIX file-system.

The definition of a snapshot [3] is "a piece of information or short description that gives an understanding of a situation of a particular time". In Git's case, a snapshot is the saved information of the top-level tracked tree and all its contained sub-trees and blobs. Since the snapshot is registered through the `git commit` command, the word 'commit' is often used to refer to a 'snapshot', which can potentially cause confusion. It is important to remember that a commit is a snapshot, a saved copy of the directory, and all its contents, at one point in time.

To summarise, the git data model is made up of trees (and subtrees) and blobs (leaves), snapshots of such trees are then saved as commits. Each of these elements are equally considered *objects* in gits internal object model, illustrated in pseudo-code below.

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | blob>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

2.2 Git's object model

"Object" is a conceptualisation of an *reference to a hash.* When these objects reference other objects, they do not refer to their location in storage, they refer to their *hash-value.* Git is a content-addressable file system, more commonly called a key-value data store. If users want to explore this system using their terminal such guides are readily available [4].

When git stores an object in its own database, it returns a hash (key) for that stored object. The hash itself is a SHA-1 cryptographic hash function of 40 characters. SHA-1 has deprecated and is considered broken by the National Institute of Standards and Technology [9]. The git development team currently has the objective of transitioning to SHA-256 [7].

Trees, blobs and commits are all objects referred to by their hash. This is unique as per each object, and *in addition it is unique with respect to each version of that object.* Blobs, which are the simplest unit, are a good example of this, showcased in the snippet down below [4]:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

The git 'objects/' directory creates sub-directories (trees) for the first two characters of each hash. As displayed above, each version of the blob test.txt are both saved in the git database with each their own unique hash, since the hash is based on the content of the blob. If removed, each version of the blob could be restored from the git database through its hash. If the same version of a file were to be stored twice, it would still yield the same identical saved blob and hash, as the hash is based on the content of the file. Example below:

```
$ echo "123" > sample.txt
$ git hash-object -w sample.txt
190a18037c64c43e6b11489df4bf0b9eb6d2c9bf
$ cp sample.txt sample2.txt
$ git hash-object -w sample2.txt
190a18037c64c43e6b11489df4bf0b9eb6d2c9bf
```

All data elements are git objects referred to by their hash, yet their functionality as objects differs.

2.2.1 Tree objects

A tree object can contain one or more hash-entries of a blob or a subtree, reflecting the structure similar to that of a UNIX file system. That is, if one were to look into the content of a tree containing a tree object, and open the content of that sub tree, the sub tree would contain its own list of hashes, showcased below [4]:

```
 $ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib

$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      blobinsubtree.rb
```

2.2.2 Commit objects

The commit object is the final of the three objects. As per the pseudocode in previous section, the commit object contains the reference to the *particular version* (a snapshot) of a tree in addition to the metadata of the author information, and a message, but most importantly, it also contains a pointer: the hash references to its *parents* - the commit or commits that preceded it (Figure 2):

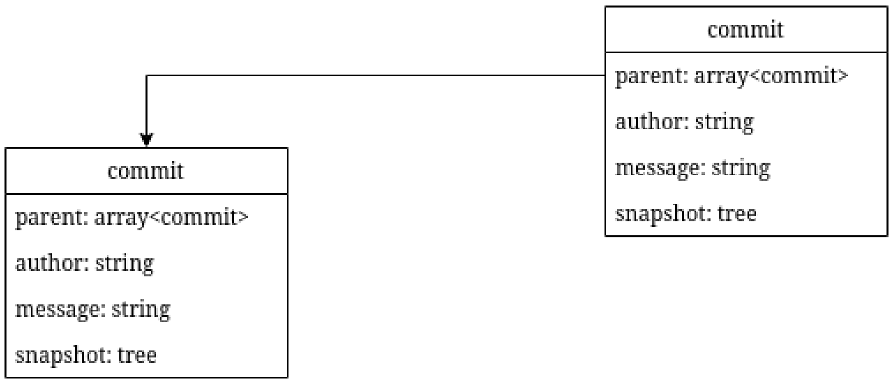

Figure 2: An example of two commits (nodes) in the DAG

This parent's functionality creates the DAG structure of git. A commit is a node. The arrows illustrated in Figure 1 of the DAG portray the parent(s)-children relationship, the arrow points towards the previously committed parent - the flow of history is 'backwards'. Through merging different *branches*, multiple parents are possible.

2.2.3 Branches

Git itself keeps track of commits through a pointer that is updated to point towards the latest commit object in the git references. However, a 40 character hash is not suitable for personal user memory. *Git references* links hashes to strings for easier memorisation [5]:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
```

The pointer for the latest commit is a branch. When a new branch is created, a new pointer is created that points at the same object. The default branch, 'master', is just the default pointer.

To keep track of which branch (pointer) is currently being used, git uses another 'special' pointer. This is what git calls the HEAD, which is also contained in the git references. This pointer structure yields some interesting functionality. Because HEAD is just another pointer, it is in fact possible to switch the HEAD to point towards a specific commit object that is currently not pointed at by any branch, a 'detached' head [5]. Creating another commit from this specific commit would then create a commit history with no branch pointer pointing at it, and would be deleted by the Git garbage collection process unless given a reference, because what is not referenced directly or indirectly is eventually discarded. In essence, when git merges two branches, it merges two commit-objects; it does a three-way merge between the two referenced commit-objects and their last common ancestor. A new commit-object is created from the merged files, and the branch pointer that was pointed to by the HEAD is updated to point towards the new commit-object. Normally, the other merged branch pointer is deleted.

2.3 Git tags and remote references

The final object are the tags [5]. Tags come in the lightweight type, which is simply a reference which never moves. An annotated tag is much like a commit-object, yet instead of a pointer towards a tree, here the pointer is free to point to any object. Usually, though it points towards a commit-object and contains metadata such as name, tagger and message.

The final reference type is the remote reference, which supports remote functionality (hosted on a platform such as GitHub). It tracks the hash of the object last pushed for each branch pointer in the remote reference. Remotes are read only as the HEAD will not allow itself to point at one, so new commit-objects can be created there.

3 Discussion: benefits and disadvantages

Diomidis Spinellis wrote as early as 2012 in his paper [13] that an important difference between Git and its predecessors such as Bitkeeper and CVS is that it 'elevates software revisions to first class citizens'. That is, the fact that every commit object is hashed and immutably saved in its entirety. This means that developers can easily revision their build, which is extremely useful in development and error-solving. Secondly, the branch-pointer functionality combined with remote repositories enables a great variety of *workflows.*

It goes without saying that, as a version control developer tool, git has been extremely successful. In 2010, git accounted for only about 10% of all repositories [8]. Today, it's more than 72% according to Openhubs data [10], its only competitor being Subversion with 23%.

3.1 Distributed vs Centralised Version Control

Contrary to Git which is a distributed version control system, Subversion is a centralised version control system [12]. A brief comparison between the two implies, in summary: Subversion is centralised in the sense that the full version of the build is only kept on the central server. Developers can work on files without keeping the full build on their local system, committing their changes to the central server.

This centralised approached requires that every unit maintains a connection to the server, which could require a lot of network traffic. However, if the central server malfunctioned, development and changes to the build would have to halt until the error is resolved. With Git, individual contributors are able to work offline, as they maintain the full build on their own system. However, this is arguably the biggest drawback of the Git system, since when the build is large and many versions are to be maintained, the total occupied disk storage grows quickly. The issue is so fundamental that Git has a new open-source extension for working with large files, Git Large-Files Storage [6]. A centralised distribution avoids this by only maintaining the full build on the server. Still, git is the more used system of the two measured in total repositories, perhaps because smaller projects outnumber projects larger in size.

3.2 Critique of the design of git

The other more significant issue with Git is the drawback of its free-form DAG design - the potential for large complexity. As Rosso and Diago noted as early as 2013 [11]:

> ”Despite its widespread adoption, Git puzzles even experienced developers and is not regarded as easy to use.”

Rosso's and Diago's paper is very recommendable, thorough walk-through of the problems that many users experience with git. They relate the issues to the violation of 3 design principles for functions forwarded by Brooks & Frederick [1], listed below with the argued violations by Rosso and Diago below:

- orthogonality – that individual functions should be independent of one another.
  Staging and committing are not orthogonal, sometimes even depending on the arguments.
- propriety – that a product should have only the functions essential to its purpose and no more;
  It is argued that staging offers inessential functionality and greatly complicates git. The weight of this argument is very subjective, some developers appreciate staging while others do not.
- generality – that a single function should be usable in many ways.
  Rosso and Diago notes that branch functionality only includes committed versions of files, not working version, but both are part of development. As such, branching breaks generality by not including the both. Additionally, if a user wants to untrack a committed file, that cannot be done with the usual staging command but requires a unique command to mark it ”assume unchanged”, and such files are not listed among the untracked files but require another unique command to be listed:

  ```
  git update-index
  git ls-files
  ```

  The critique of generality is thus that some functionality do not do enough for efficiency.

The critique of Rosso and Diago focuses primarily on the design of git functionality, especially staging, and could be done with a similar distributed system without such functionality such as *Gitless*, which the two promote. It should be noted that the git object model that is a fundamental for the functionality is irrespective of this critique of git functions as violators of good design and overtly complex. It only *enables* the arguably overt complexity. Furthermore, some developers enjoy the staging functionality.

Notably, Diago and Rosso's papers are perhaps the only academic critique of the hegemonic version control system. There is surprisingly little academic work on version control and future work in this technology should be encouraged.

4 Conclusion

Git encapsulates snapshots of directories as immutable commit-objects and saves them in a key-value hash store. The key-value hash store enables every object to be referred to by its hash reference. Git uses pointer-functionality to interrelate these stored objects and enable non-linear workflows. In combination, git provides efficient, distributed version control which treats old versions as first-class members and enables powerful software revision, an advantage that explains its huge success. The structure is not without disadvantages however, as the fully saved directories can require large quantities of storage and its structure cause excessive complexity on larger projects.

References

[1] Frederick P Brooks Jr. *The design of design: Essays from a computer scientist.* Pearson Education, 2010.

[2] Zack Brown. *A Git Origin Story.* 2018. URL: `https://www.linuxjournal.com/content/git-origin-story` (visited on 04/05/2021).

[3] *Cambridge Dictionary.* 2021. URL: `https://dictionary.cambridge.org/dictionary/english/snapshot` (visited on 04/07/2021).

[4] *Git Internals - Git object.* 2021. URL: `https://git-scm.com/book/en/v2/Git-Internals-Git-Objects` (visited on 04/07/2021).

[5] *Git Internals - Git References.* 2021. URL: `https://git-scm.com/book/en/v2/Git-Internals-Git-References` (visited on 04/08/2021).

[6] *Git Large File Storage.* 2021. URL: `https://git-lfs.github.com/`.

[7] *Git Transition Objective.* 2021. URL: `https://git-scm.com/docs/hash-function-transition/2.23.0` (visited on 04/07/2021).

[8] *Internet Archive Wayback Machine.* 2021. URL: `http://web.archive.org/web/20100821122603/http://www.ohloh.net/repositories/compare` (visited on 04/12/2021).

[9] *NIST Policy on Hash Functions.* 2021. URL: `https://csrc.nist.gov/projects/hash-functions/nist-policy-on-hash-functions` (visited on 04/07/2021).

[10] *Openhub Repositories Comparison.* 2021. URL: `https://www.openhub.net/repositories/compare` (visited on 04/12/2021).

[11] Santiago Perez De Rosso and Daniel Jackson. "What's Wrong with Git? A Conceptual Design Analysis". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software.* Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 37–52. ISBN: 9781450324724. DOI: `10.1145/2509578.2509584`. URL: `https://doi.org/10.1145/2509578.2509584`.

[12] C Michael Pilato, Ben Collins-Sussman, and Brian W Fitzpatrick. *Version control with subversion: next generation open source version control.* " O'Reilly Media, Inc.", 2008.

[13] Diomidis Spinellis. "Git". In: *IEEE Software* 29 (3 2012), pp. 100–101. DOI: `10.1109/MS.2012.61`.

[14] *Version Control*. 2021. URL: `https://missing.csail.mit.edu/2020/version-control/` (visited on 04/05/2021).