Merkle DAG

Angela Merkel (source)

Not invented by the German chancellor, but interesting nonetheless.

The nuance of the naming might not stick the first time; there is a distinction between a Merkle tree typically used to incrementally verify data integrity and a Merkle DAG as used by Git. The key difference between the two structures is that (in the more-general DAG one) any node can potentially hold data and references as well as a hash of its parents.

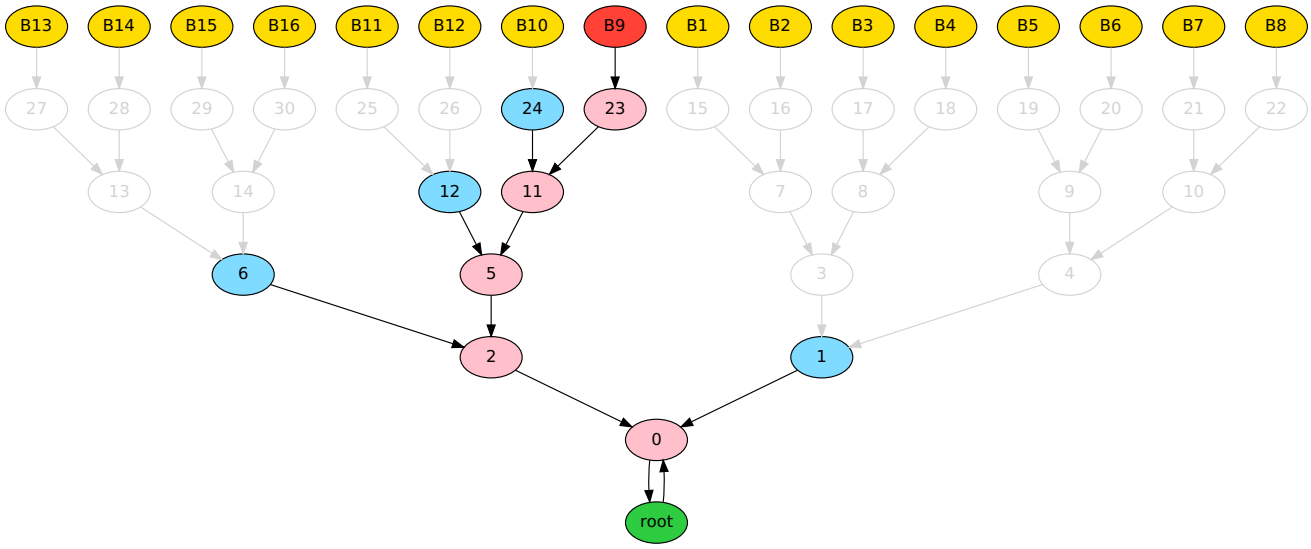
This is true of the graph of objects making up the the "main" DAG in everyone's Git repos, except that as far as my understanding goes Git's Merkle DAG is used more for high performance addressing and perhaps deduplication rather than ensuring data integrity between peers.

Merkle tree

Let's look at how a binary Merkle tree is used in peer-to-peer systems to allow verification of untrusted data with a high degree of confidence and low metadata overhead.

To be able to verify the hash of a block I have, I need a trusted root hash (a trusted hash of all blocks) and a series of hashes that can be used to reconstruct the hash tree for the blocks I don't have. Because the hashes are in a tree structure, this doesn't necessarily mean the hashes of all the missing blocks (although that would work too).

In the example below, we have been sent block `B9` (red) and the required "uncle" hashes (blue) by an untrusted peer. We don't have any other verified blocks (yellow) and we need to verify the integrity of the block we've been sent. To do this we don't need anything apart from the hash of the untrusted block (23), the untrusted "uncle" hashes and the trusted root hash (green). Calculating the missing nodes in the Merkle tree (pink) by hashing the descendants will eventually yield an untrusted hash representing all the blocks. This untrusted root hash can then be compared to our trusted root hash to decide whether to keep the block or not (and treat that peer as untrustworthy in the future, etc).



The efficiency comes from peers not needing to know so much. In fact, there was quite a lot we didn't need to know (all the grey stuff in the diagram). The hashes that make up the tree for the blocks we don't yet have can remain unknown because those nodes in the tree are covered by the blue "uncle" nodes.

Cool.

Git's DAG

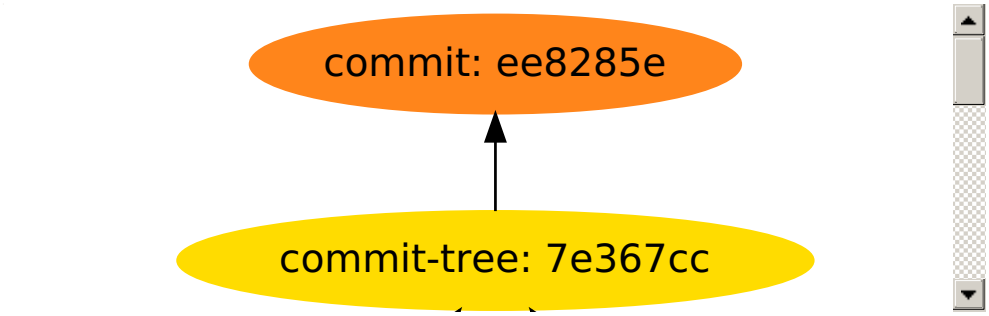
So we briefly covered the application of a Merkle tree in the context of verifying untrusted blocks of files in peer-to-peer systems, but how does that apply when we're talking about Git?

Reading Tommi Virtanen's great article Git for computer scientists is a good place to start seeing how Git's DAG works. Taking (some) inspiration from git-big-picture I whipped up a script to examine Git's DAG. We can see how deduplication is handled by addressing content rather than files. Follow along with my experiment:

```
$ git init
$ mkdir A
$ touch A/a A/b
$ touch B
$ tree .
.
├── A
│   ├── a
│   └── b
└── B

1 directory, 3 files
$ git add .
$ git commit -m 'init'
$ git rev-parse HEAD | cut -c 1-7
ee8285e
```

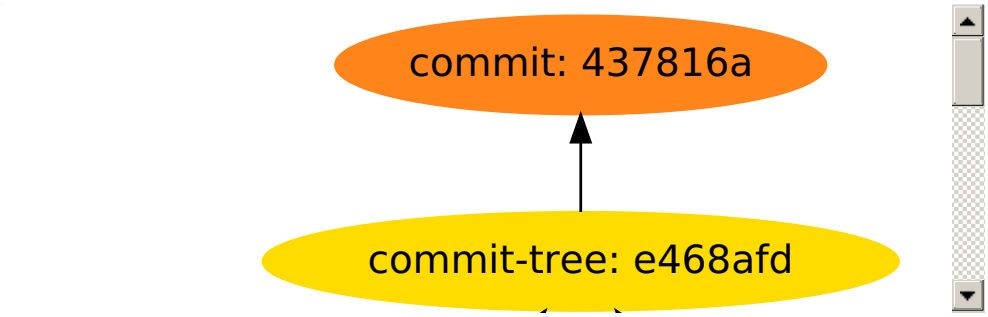
Ok, so that's the simplest repo known to man and we have the revision ID of `HEAD`. Let's look at the what's going on under the hood.



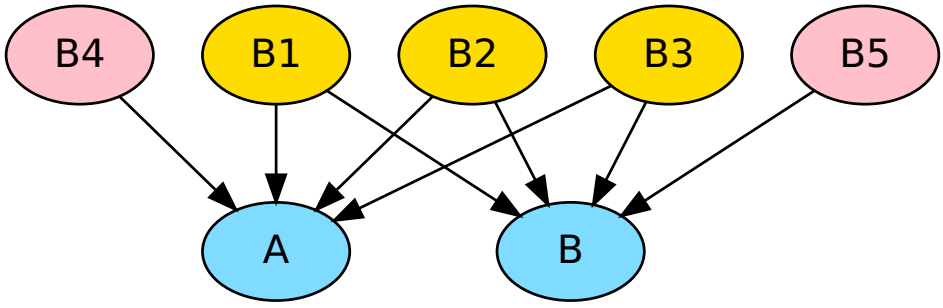
We can see the directory `A` (or tree object `296e560`) and the files `B`, `a` and `b` we created. Notice that all the files reference that same blob object `e69de29`, that's because they are all empty files (and therefore have the same content, nothing). If we alter file `a` to not be empty (and therefore have different content) like this:

```
$ echo 'hello' > A/a
$ git add A/a
$ git commit -m 'altered a'
$ git rev-parse HEAD | cut -c 1-7
437816a
```

Not only do we get a new commit ID and commit-tree ID (`e468afd`), but we also see the underlying DAG change:



Both files `b` and `B` :smile: still share a blob, but `a` now has a blob of its very own. This also demonstrates that in Git's model, blob objects correspond to one-to-one with files (sans directory location) which works fine if you only want to deduplicate files that have exactly the same content, but deduplication could be more aggressive if files were split into blocks and deduplicated at block-level instead of file-level.



Files `A` and `B` share most blocks (yellow), so blocks 1-3 are used by both, only blocks 4 and 5 (pink) are unique to the individual files.

That's enough colourful graphs for now. Get back to work.

0b5fbcc

previous The ipfs

JavaScript Framework Rochambeau next