**3. Design**

A version control system design is focused around three core functionalities:

a. Storing Content

b. Tracking changes to the content

c. Distributing the content and history with collaborators

While maintaining these features and functionalities of other version control systems, Git was developed with other three important design goals:

a. Support distributed workflows

b. Offer safeguard against content corruption

c. Offer high performance

These goals have been accomplished and maintained by use of directed acyclic graphs (DAGs) for content storage, reference pointers for heads, object model representation, remote protocol, and merging of trees. DAGs are the directed graphs with no directed cycles. That is, starting from an edge, there is no sequence of directed vertices that can be followed to reach the initial edge. These are used for space-efficient representation of collection of sequences.

A Directed Acyclic Graph (DAG)

**3.1 Content Storage**

The common design choices for storing content in a version control system are either delta-based changeset or directed acyclic graph content representation. Delta-based changeset maintains difference between two versions of the content, along with metadata. Following figure illustrates this concept:

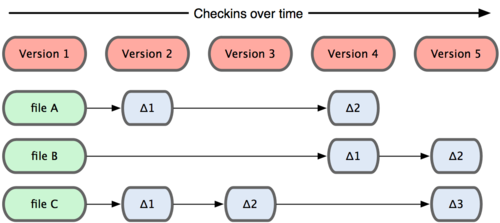


Figure 1. Store data as changes to base version

Git on the other hand, employs DAGs for content storage. A reference is maintained for every commit. The edges in the graph represent objects that form the DAGs inside the repository. If files are not changes, a link to previous version of the file is maintained. Following figure illustrates this concept:

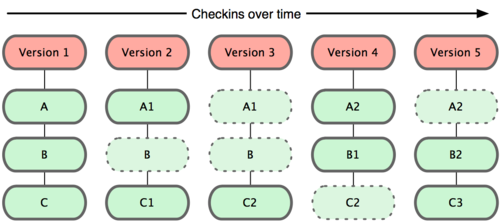


Figure 2. Snapshots stored over a period of time

**3.2 Commit and Merge Histories**

To maintain the history and change-tracking, version control systems use one of the following approaches:

a. Linear history

b. Directed acyclic graph (DAG)

Git again uses DAG for this purpose. Every commit has metadata about its ancestors. Using DAG allows branching and merging of history cases efficiently. As new commit objects are linked to their ancestors who in turn are linked to their ancestors, they form a ‘history’.

HEAD

Circles are commit objects, which link to one or more parent commits – back to their original ancestor(s) – thus forming a history

Every commit holds a tree, and every tree may contain any number of other trees and blobs in its leaves

Figure 3. Commit and Merge history

The following figure illustrates how DAGs are used to create the commit and history patterns. Initially, the object ‘98ca9’ was stored in the file system. After changes, a new commit was stored as ‘34ac2’. This object has the reference of its ancestor ‘98ca9’. This creates a history flow that can be traced to access ancestor objects. Also note that, every object has description about the object itself, the author and committer.

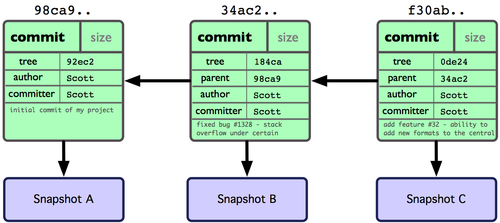


Figure 4. Git object data for multiple commits

**3.3 Distribution**

Version control systems provide content distribution among the collaborators in three different ways:

a. Local-only: No distribution of content and history among collaborators.

b. Central Server: all the changes to the repository must transact through a central server to maintain history.

c. Distributed model: a publicly accessible repository where collaborators ‘push’ their work. Collaborators can commit locally and push later thereby allowing offline work.

When a collaborator commits locally, a new object is created containing all the changes and the original content. For every directory structure above the changed file, a new object is created. These newly created objects form a new DAG. At this point the changes are local. Once the collaborator decides to ‘push’ the changes, this DAG is sent to the public repository.

**3.4 Data Structures**

Git has two data structures:

**3.4.1. Object Database:**

The object database contains four types of objects:

a. Blob:

A blob is the content of a file. It has no other information like file name, time stamps or metadata. It is the fundamental unit of Git. Blobs are used as a leaf node for trees.

b. Tree:

A tree is the snapshot of original tree in the repository. It contains a list of file names, symbolic link, or directory’s content.

c. Commit:

A commit object links trees together to create history. It contains the tree object, time stamp, log message, and the names of zero or more parent commits.

d. Tag:

A tag object is a container that contains reference to another object and can hold additional metadata related to another object.

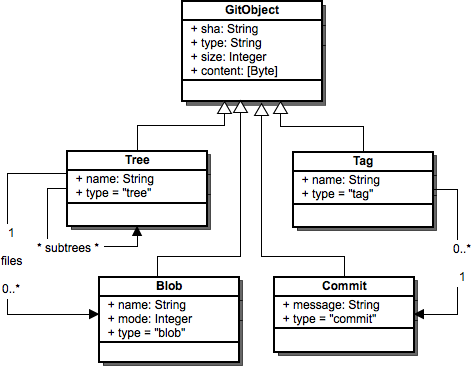


Figure 5. Git Objects

All object primitives are referenced by a SHA (a cryptographic hash function), a 40-digit object identity with following properties:

* Identical objects will have same SHA.
* Different objects will have different SHA.
* Recalculating SHA will identify any partial copy or data corruption.

The first two properties enable Git to provide distributed model of version control. The third property allows Git to safeguard data against data corruption.

**3.4.2 Index**

The index serves as a connection point between object database and the working tree. It refers to the set of newly created trees and blobs. These new objects are bound into a new tree but not committed. Until they are committed to the repository, they are referenced by the index. This allows the collaborator to makes changes locally. Only when he/she is sure of the changes, the tree can be pushed into the repository.

NEW HEAD

Index

HEAD

The state of index becomes the tree of the next commit

**4. Implementation**