Git Architecture

Git's Origin

Git was born in 2005 out of the needs and frustrations of the Linux kernel developers. At that time, the kernel was managed using two VCSs: BitKeeper and CVS.
When BitMover, the maker of BitKeeper, announced it would revoke licenses for some Linux kernel developers, Linus Torvalds quickly began developing Git.

Torvalds' main philosophical goal for Git was to be the "anti-CVS," with three specific usability design goals:

1. Support distributed workflows similar to BitKeeper.
2. Provide safeguards against content corruption.
3. Ensure high performance.

These goals have been achieved through features like directed acyclic graphs (DAGs) for content storage, reference pointers for heads, object model representation, and a robust remote protocol. Git also supports advanced tree merging capabilities.

Version Control System Design

Most VCSs have three primary functions:
1. Storing Content
2. Tracking Changes to the Content
3. Distributing the Content and History with Collaborators

1. Storing Content :
Version Control Systems (VCS) commonly store content using either delta-based changesets or directed acyclic graph (DAG) representations:

a. Delta-based Changesets: Capture differences between two versions of content along with metadata. This approach is efficient in terms of storage but can be complex to manage during merges and retrievals.

b. DAG Content Representation: Involves objects forming a hierarchical structure that mirrors the filesystem tree, capturing snapshots of the content at each commit. Git uses DAGs with various object types stored in its "Object Database."

2. Commit and Merge Histories:

VCS tools track history using either:

- Linear History: Simpler but less flexible, commonly seen in systems like Subversion.
- DAG for History: Git uses a DAG to track history, where each commit includes metadata about its parent commits. This allows Git to support complex branching and merging scenarios, making it more powerful than linear history systems.

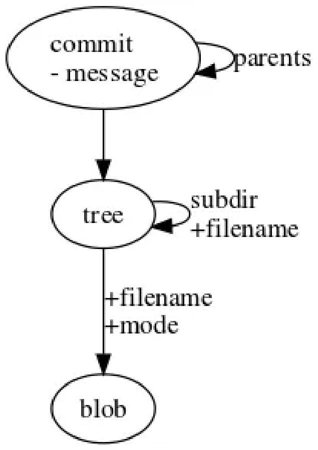
3. Distribution Models in VCS

VCS solutions distribute content using one of three models:

- Local-only: Suitable for standalone development without collaboration.
- Central Server: Used by systems like Subversion, where a central repository manages all changes.
- Distributed Model: Used by Git, where each collaborator has a full copy of the repository, allowing offline work and local commits before sharing changes.

Git Architecture

Object Database



Git utilizes this Directed Acyclic Graph structure for content storage. Git is essentially a content-addressable file system made up of objects that form a hierarchy which mirrors the content's filesystem tree.

Git uses four basic primitive objects for content storage in the local repository:

1. **Blob:** Represents a file stored in the repository. It is identified by a SHA-1 hash of its contents. Blobs store file data but do not include metadata like filename.
2. **Tree:** Represents a directory listing and points to blobs or other trees. Each tree object is identified by a SHA-1 hash of its content (which includes references to blobs and subtrees).
3. **Commit:** Represents a snapshot of the repository at a specific point in time. It points to a tree object that represents the top-level directory for that commit. Commits also contain metadata such as author, committer, commit message, and parent commits. The SHA-1 hash of a commit uniquely identifies it and includes all these attributes.
4. **Tag:** A reference to a specific commit. Tags include a name and point to a commit object in the repository history. They are used to mark specific points in history such as release points.

Objects are immutable and identified by their SHA-1 hashes, ensuring data integrity and enabling efficient content sharing and storage across repositories. The use of hashes ensures that identical content will have the same hash, while any changes or corruption will result in a different hash, providing built-in integrity checks and safeguards against data corruption.

Merge Histories in Git

In contrast to linear-history VCSs like Subversion, Git employs a DAG-based merge history approach, which allows for more flexible and accurate management of parallel development branches and their merge histories.

Git uses a Directed Acyclic Graph (DAG) to manage merge histories. Each commit points to its parent commit(s), enabling Git to track the complete history of merges and understand which changes have been propagated between branches. This is achieved through the parent relationships in commit objects, identified by their unique SHA hashes.
Git's merge commits explicitly record which commits from other branches were merged, making it clear which changes are integrated into which branches.

Distribution model of Git

Git operates on a distributed model where each collaborator maintains a complete copy of the repository locally.Git allows collaborators to work offline, independently making changes and commits to their local repository. Each collaborator has a full history and version of the project, including all branches and commits. Changes are shared among collaborators by pushing them to a remote repository. Other collaborators can then pull these changes to update their local copies.

To create a new Git repository locally, you use the command ***git init***. This command initializes a new repository in the current working directory, creating a ***.git*** directory where Git stores all necessary files and metadata.

.git Directory Structure :

```
.git/
|-- HEAD
|-- config
|-- description
|-- hooks
|   |-- applypatch-msg.sample
|   |-- commit-msg.sample
|   |-- post-commit.sample
|   |-- post-receive.sample
|   |-- post-update.sample
|   |-- pre-applypatch.sample
|   |-- pre-commit.sample
|   |-- pre-rebase.sample
|   |-- prepare-commit-msg.sample
|   |-- update.sample
|-- info
|   |-- exclude
|-- objects
|   |-- info
|   |-- pack
|-- refs
    |-- heads
    |-- tags
```

1. .git/HEAD: This file points to the current branch reference (e.g., refs/heads/master), indicating the branch where new commits will be made.
2. .git/config, .git/description, .git/info/exclude: Configuration files that control various aspects of the repository.
3. .git/hooks/: Directory containing scripts that execute on specific Git lifecycle events.
4. .git/index: The Git index, not shown directly but mentioned, acts as a staging area where changes are prepared before committing.

5. .git/objects/: Directory where Git stores all objects (commits, trees, blobs) in its object database. Objects are identified by their SHA-1 hashes and are immutable once created.
6. .git/refs/: Directory storing reference pointers to commits or tags. For example, .git/refs/heads/ contains references to local branches like master.