A version control system (VCS) generally needs to do three main things: store files and their content, keep a record of changes to those files (including information on merges), and share the files and their history with others. However, not all VCSs need to support sharing with others as a core function. By looking at different VCS options besides Git, we can better understand the design decisions that went into creating Git.

GIT ARCHITECTURE

Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Its architecture revolves around a few core concepts: repositories, commits, branches, and distributed workflows. At the heart of Git is the repository, a database storing all the project's history. Each repository consists of snapshots, or commits, which record changes made to the project files over time. Commits are identified by unique SHA-1 hash codes, ensuring integrity and traceability. Branches in Git allow for parallel development; each branch represents an independent line of development. This makes it easy to experiment, develop features, and fix bugs without disrupting the main codebase. Git's distributed nature means that every developer has a full copy of the entire repository history on their local machine, allowing for offline work and robust collaboration. Changes can be shared between repositories via push, pull, and fetch operations, facilitating a flexible and decentralized workflow. This architecture not only enhances performance and reliability but also empowers developers to work in a more autonomous and efficient manner.

In Git, commits are uniquely identified by SHA-1 hash codes, which are 40-character strings generated through a cryptographic hash function. This process involves taking the contents of a commit and running them through the SHA-1 algorithm, which produces a unique identifier for that specific commit.

A SHA-1 hash in Git is computed based on several pieces of information:

1. **Content of the commit:** This includes the tree object (representing the state of the directory), the parent commit(s) (linking the commit to its predecessor(s)), the commit message, the author's name, email, and timestamp, and the committer's name, email, and timestamp.
2. **Metadata:** The above elements are concatenated into a single string and then fed into the SHA-1 hashing function.

The resulting hash looks something like this: `e2adf8ae3d8e26d94a0e8d7a72f26eb1a1c6e103`. This 40-character string acts as a unique fingerprint for the commit.

This system ensures:

- **Integrity:** Any change in the commit's content or metadata will result in a completely different SHA-1 hash, making it immediately apparent if any data has been tampered with.

- **Uniqueness:** The vast space of possible SHA-1 values makes hash collisions (where two different commits end up with the same hash) extremely unlikely, ensuring each commit is uniquely identifiable.

In summary, SHA-1 hashes in Git provide a secure and efficient way to uniquely identify commits, ensuring the integrity and traceability of the project’s history.

Content Storage

In the world of version control systems (VCS), there are two common ways to store content: delta-based changesets and directed acyclic graphs (DAGs). Delta-based changesets focus on recording the differences between two versions of the same content along with some additional information. This means they store changes incrementally. On the other hand, representing content as a directed acyclic graph involves creating a hierarchy of objects that reflect the structure of the content as it exists in the filesystem at a particular point in time. Each snapshot captures the state of the entire content, reusing unchanged parts to save space. Git uses this DAG approach to store content, organizing it into different types of objects that form these graphs. This structure helps Git efficiently manage and navigate the content history. The "Object Database" section later will explain the various object types that make up these DAGs within a Git repository.

Commit and Merge

When it comes to tracking history and changes, most version control systems (VCS) use one of two methods: linear history or a directed acyclic graph (DAG) for history. Git uses a DAG to store its history. Each commit in Git includes metadata about its parent commits, and a commit can have any number of parents. For instance, the very first commit in a Git repository has no parents, while a commit resulting from a three-way merge has three parents.

A major difference between Git and systems like Subversion, which uses a linear history, is Git's robust support for branching. This branching capability allows Git to record most merge histories, providing a more comprehensive and flexible way to manage changes.

Git uses directed acyclic graphs (DAGs) to enable full branching capabilities by linking the history of a file up through its directory structure to the root directory, which is connected to a commit node. This commit node can have one or more parent commits. This structure gives Git two significant advantages over version control systems derived from RCS:

1. **Content Identity and Efficiency**: When a file or directory node in the graph has the same SHA reference in different commits, Git guarantees that these nodes contain the same content. This allows Git to avoid redundant comparisons, making content diffing more efficient.
2. **Efficient Merging**: When merging two branches, Git merges the content of two nodes in the DAG. The DAG structure helps Git quickly and efficiently find common ancestors, making the merge process more effective compared to the RCS family of VCS.

These properties allow Git to handle history and content in more definite and efficient ways.

The Object Database

Git uses four basic primitive objects to build all types of content in the local repository. Each object type has attributes: type, size, and content. The primitive object types are:

1. **Tree**: Represents a directory. An element in a tree can be another tree or a blob.
2. **Blob**: Represents a file stored in the repository.
3. **Commit**: Points to a tree representing the top-level directory for that commit, along with parent commits and standard attributes.
4. **Tag**: Has a name and points to a commit at a specific point in the repository history.

All these objects are referenced by a SHA, a 40-digit identifier with the following properties:

1. **Identical Objects**: If two objects are identical, they will have the same SHA.
2. **Different Objects**: If two objects are different, they will have different SHAs.
3. **Corruption Detection**: If an object is only partially copied or corrupted, recalculating the SHA will identify the corruption.

The first two properties are crucial for Git's distributed model, ensuring that identical objects have the same identifier across different repositories. The third property helps safeguard against data corruption.

While using DAG-based storage for content and merge histories provides clear benefits, delta storage can be more space-efficient for many repositories than using loose DAG objects.