go in depth (search like a hell) to explain me how git exactly stores file in .git directory? how it uses data structure (include all technical jargons)
I'm a Software Enginner, pretend yourself as sotware engineer as well

use Elon Musk's first principals (Semantic Tree approach) to explain stuff!.

DeepSearch
59s • 10 sources

Evaluating Git efficiency
Assessing storage impact
Analyzing maintenance effects
Confirming data structures
Completed

Exploring Git storage

delta compression to store differences between objects. This keeps things tidy and efficient.

- Understanding Git's storage helps with debugging, like recovering from repo issues. It also informs strategies for managing large repos or optimizing storage.
- The function call provided a detailed breakdown of Git's data structures, such as blobs storing file content with hashes like `2e65efe2a145dda7ee51d1741299f848e5bf752e`. It also covered trees, commits, the index, and refs, aligning with my earlier findings.
- Trees structure the directory, referencing blobs and other trees, each with a SHA-1 hash in `.git/objects/`. This organizes files hierarchically, making navigation easy.
- Mary Rose Cook's Gitlet explanation is great for visualizing Git's internals. She breaks down how blobs hold file contents, trees

10 web pages

Key Points

- Git stores files in the `.git` directory using blobs for file contents, trees for directory structures, and commits for snapshots, all identified by SHA-1 hashes.
- It seems likely that this system allows efficient storage by reusing identical content across commits.
- Research suggests Git uses packfiles for further compression, which might be an unexpected detail for many users.

How Git Stores Files

Git, a distributed version control system, keeps the project history in the `.git` directory. When you add a file, Git creates a **blob** to store its content, identified by a unique SHA-1 hash. For example, a file with "hello" might be stored as a blob with hash `d670460b4b4aece5915caf5c68d12f560a9fe3e4` in `.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4`. When you commit, Git builds **trees** to represent the directory structure, referencing blobs and other trees, and creates **commits** that point to these trees, capturing the repository's state with metadata like author and timestamp.

This approach is efficient because Git reuses blobs for identical content, saving space. An unexpected detail is that Git can also use **packfiles** for compression, storing differences between objects to optimize storage over time.

For more details, check the official Git documentation on Git Objects.

A Comprehensive Survey Note on Git's File Storage in the .git Directory

This note provides an in-depth exploration of how Git stores files in the `.git` directory, focusing on the data structures and technical details. It aims to mimic the style of a professional article, offering a strict superset of the information in the direct answer section, and includes all relevant details from the research conducted.

Introduction
Git, a distributed version control system, is designed to track changes to files over time, enabling collaboration, version reversion, and branch management. The `.git` directory, hidden within the project folder, serves as the repository's backbone, storing the entire history and state of the project. This survey note delves into the mechanics of how Git stores files, using a content-addressable filesystem and leveraging specific data structures like blobs, trees, and commits.

Core Data Structures and Storage Mechanisms
Git's storage model is built on three primary data structures, each serving a distinct purpose:

- **Blobs (Binary Large Objects):**
  - **Definition:** Blobs store the raw content of files, without any metadata like filenames or paths.
  - **Storage**: Each blob is identified by a SHA-1 hash of its content, ensuring content-addressable storage. The blob is stored in `.git/objects/`, with the first two characters of the hash forming a subdirectory and the remaining 38 characters forming the filename. For instance, a file with content "hello" might have a hash `d670460b4b4aece5915caf5c68d12f560a9fe3e4`, stored in `.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4`.
  - **Efficiency**: Since blobs are based on content, identical files (even with different names or locations) share the same blob, enabling deduplication and space efficiency.
  - **Creation**: Blobs are created when files are added using `git add`, and their hashes are recorded in the index (staging area).
- **Trees**:
  - **Definition:** Trees represent the directory structure, acting as containers for blobs (files) and other trees (subdirectories).
  - **Structure**: Each tree entry includes:
    - Mode (permissions, e.g., `100644` for a regular file, `040000` for a directory).
    - Type (blob or tree).
    - SHA-1 hash of the referenced blob or subtree.
    - Name of the file or directory.
  - **Storage**: Trees are also stored in `.git/objects/`, identified by their SHA-1 hashes, similar to blobs.
  - **Example**: A root tree might reference a blob for `README.md` and a subtree for a `src` directory, organizing the hierarchical structure.
- **Commits**:
  - **Definition**: Commits capture snapshots of the entire repository at specific points in time, forming the history graph.
  - **Structure**: A commit object includes:
    - The SHA-1 hash of the root tree it points to.
    - SHA-1 hash(es) of parent commit(s), enabling history tracking (multiple parents for merge commits).
    - Author and committer information (name, email, timestamp).
    - Commit message for documentation.
  - **Storage**: Commits are stored in `.git/objects/`, with their SHA-1 hashes determining their location.
  - **Graph Property**: Commits form a directed acyclic graph (DAG), where each commit points to its parent(s), supporting branching and merging.

Storage Process and Workflow
The process of storing files in Git involves several steps, reflecting its operational workflow:

1. **Adding Files (`git add`):**
   - When a file is added, Git reads its content, computes a SHA-1 hash, and stores it as a blob in `.git/objects/`.
   - The blob's hash is recorded in the **index** (`.git/index`), which maps file paths to their corresponding blob hashes. This staging area prepares files for commit.
2. **Committing Changes (`git commit`):**
   - Git creates a **tree object** representing the current directory structure, using the blobs from the index. This tree recursively references other trees for subdirectories.
   - A **commit object** is then created, pointing to the root tree, including metadata (author, timestamp, message), and referencing the previous commit (parent).
   - Both the tree and commit are stored in `.git/objects/`, identified by their SHA-1 hashes.
3. **Efficiency Through Reuse**:
   - Git's content-addressable design ensures **content deduplication**: identical file contents are stored as a single blob, reducing redundancy.
   - **Tree reuse** occurs when directories remain unchanged between commits, allowing Git to reference existing trees.
   - Over time, Git can optimize storage by creating **packfiles**, which compress multiple objects into a single file using **delta compression**. This stores differences between objects (e.g., diffs between file versions) rather than full copies, enhancing efficiency for large repositories.

References and Navigation
Git uses references to navigate and manage the repository, complementing the object storage:

- **HEAD**:
  - A special reference pointing to the current commit, usually through a branch. Stored in `.git/HEAD`, it might contain `ref: refs/heads/main`.
- **Branches**:
  - Stored in `.git/refs/heads/`, each branch is a file containing the SHA-1 hash of the commit it points to. For example, `.git/refs/heads/main` might contain `7f54a437d87cd1f241cfb893c4823bc7e60c19ec`.
- **Tags**:
  - Similar to branches, stored in `.git/refs/tags/`, used to mark specific points in history, such as releases.

These references allow developers to switch between different states (e.g., `git checkout main`) without directly manipulating SHA-1 hashes, enhancing usability.

Practical Implications for Software Engineers
Understanding Git's internal storage model is crucial for advanced usage and troubleshooting:

- **Debugging:** If a repository becomes corrupted, knowing where data is stored (e.g., `.git/objects/`) can help recover lost commits or identify issues.
- **Performance Optimization**: Large repositories benefit from Git's deduplication and packfiles, but engineers can optimize workflows by understanding these mechanisms, such as using `git gc` to pack objects.
- **Advanced Operations**: Concepts like rebasing, merging, and resetting are easier to grasp when you understand how commits, trees, and blobs interact, forming the DAG.

Example Workflow
To illustrate, consider a simple repository with one file, `hello.txt`, containing "hello":

- Run `git add hello.txt`:
  - Creates a blob with hash `d670460b4b4aece5915caf5c68d12f560a9fe3e4`, stored in `.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4`.
  - Index records: `hello.txt d670460b4b4aece5915caf5c68d12f560a9fe3e4`.
- Run `git commit -m "Initial commit"`:
  - Creates a tree referencing the blob, with hash, say, `f2a3b4c5d6e7f8g9h0i1j2k3l4m5n6o7p`.
  - Creates a commit pointing to this tree, with hash, say, `a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6`.
  - Branch `main` (`.git/refs/heads/main`) points to this commit.
- Modify `hello.txt` to "hello again" and commit:
  - Creates a new blob with hash `e69de29bb2d1d6434b8b29ae775ad8c2e48c5391`.
  - Creates a new tree and commit, referencing the previous commit as parent.

The old blob remains in `.git/objects/`, demonstrating Git's reuse of unchanged content.

Tables for Clarity
To organize the data structures and their properties, consider the following tables:

Object Type	Description	Storage Location	Example Hash
Blob	Stores file content, no filename, used for version control of file contents.	`.git/objects` (e.g., `d6/70460b...`)	`d670460b4b4aece5915caf5c68d12f560a9fe3e4`
Tree	Stores directory structure, contains entries with SHA-1 of blobs or subtrees.	`.git/objects` (e.g., `fd/4f9947...`)	`f2a3b4c5d6e7f8g9h0i1j2k3l4m5n6o7p`
Commit	Stores snapshot metadata, points to top-level tree, parent commits, author info.	`.git/objects` (e.g., `a1/b2c3d4...`)	`a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6`

Reference Type	Description	Storage Location	Example
HEAD	Points to current commit, usually via branch.	`.git/HEAD`	`ref: refs/heads/main`
Branch	Points to a commit, mutable.	`.git/refs/heads/` (e.g., `main`)	`7f54a437d87cd1f241cfb893c4823bc7e60c19ec`
Tag	Marks specific commit, often for releases.	`.git/refs/tags/`	N/A (varies by tag name)

These tables summarize the key components and their roles, enhancing understanding of Git's storage model.

Conclusion
Git's storage in the `.git` directory is a sophisticated system based on immutable objects (blobs, trees, commits) identified by SHA-1 hashes, organized to allow efficient storage, retrieval, and manipulation of project history. This design supports collaboration, version control, and performance optimization, making Git a powerful tool for software development.

Key Citations

- Git Internals Git Objects detailed explanation
- How Git stores data comprehensive guide
- Git from the inside out detailed walkthrough