READ MODE · ANSWER FIRST

How Git works under the hood: objects, trees, commits, refs and packfiles

See how Git serializes blobs, builds tree objects from the index, links commits and refs, walks reachability and stores packfiles.

CHEAT SHEET · 6 ESSENTIAL IDEAS

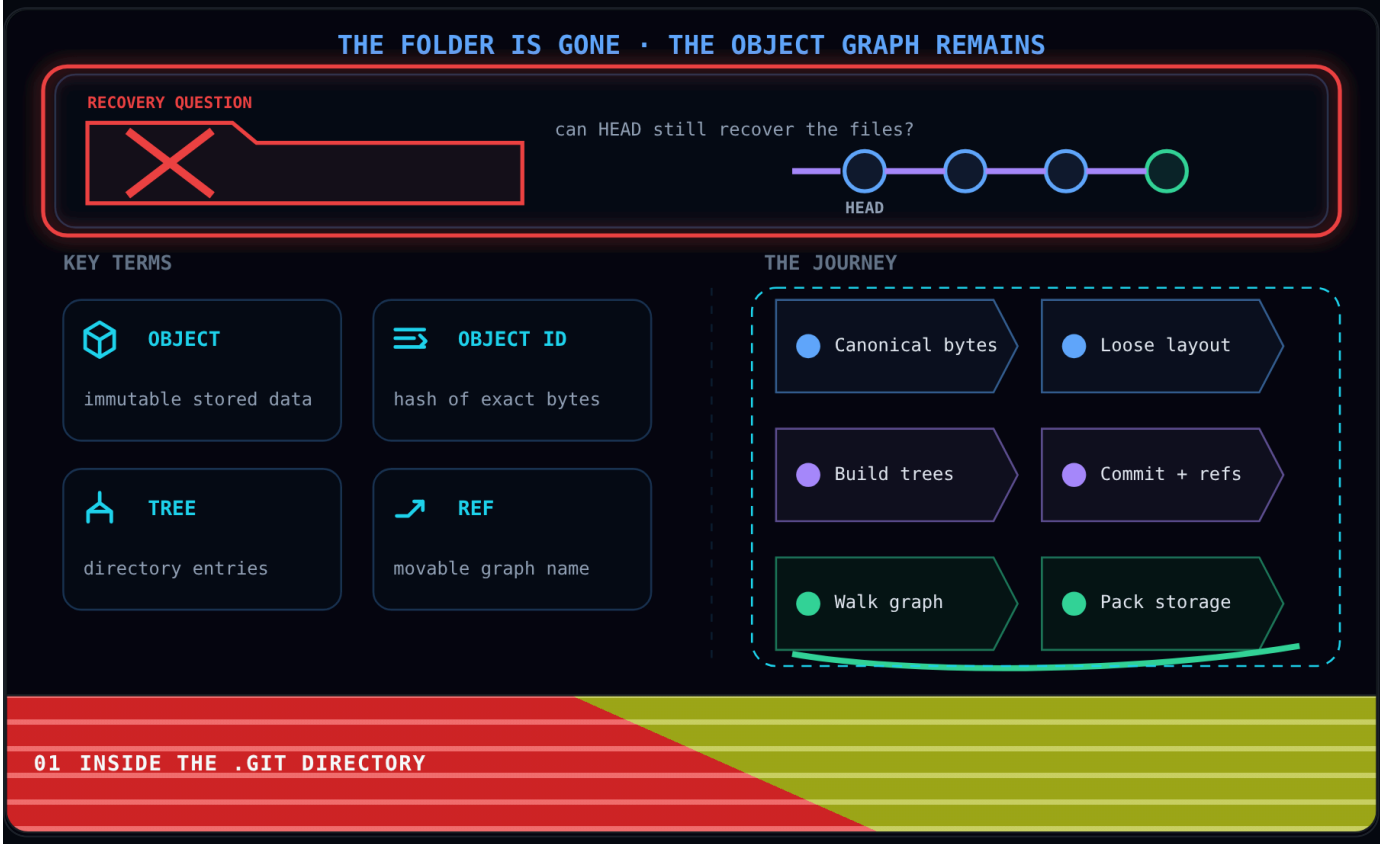
The whole story in 6 lines

Git is an immutable content-addressed object graph whose movable refs choose which snapshot history names today.

1. Git hashes a type and length header with content, so identity covers both meaning and exact bytes.
2. A loose object uses its first hash byte as a directory shard and stores compressed canonical bytes.
3. Git folds sorted index paths into bottom-up tree objects whose entries hold raw object IDs.
4. A commit names one root tree and its parent commits, while refs provide movable names for graph tips.
5. Reading a path means following object pointers, and unchanged blobs can serve many snapshots.
6. Packfiles change physical storage with compression and deltas but preserve logical object identity.

STAGE 1 OF 8

Inside the .git Directory



Imagine opening your project and finding every tracked file gone. The hidden `.git` directory still survives, so Git can reconstruct the last committed version. To understand how, we need to look beneath familiar commands and find what Git actually saved.

Git calls each stored piece of data an object. An object ID is the permanent name calculated from that data. Tree objects describe folders, while refs are movable names that help people find important points in history.

We will follow one small JavaScript file through the entire system. First its bytes become an object. Then trees give that object a path, a commit records the snapshot and refs make the commit easy to find. Finally, Git reorganizes storage without changing identity.

Here is the idea to carry forward: Git stores stable objects first, then connects them with pointers. The working folder is only one version materialized from that deeper graph. Let us begin with how ordinary file bytes receive a permanent object ID.

STAGE 2 OF 8

Canonical Object Bytes

★ If you remember one thing · Changing one content byte creates a different immutable blob identity while the original object remains addressable.

Try it in Watch mode. Make the second file reuse the existing blob object.

02 CANONICAL OBJECT BYTES

We start with the familiar file `src/app.js`. Git stores its exact UTF-8 content as a blob, including the final newline, but the path `src/app.js` is not part of that blob. Folder names will enter the story later through tree objects.

You might expect Git to hash only the file content. Instead, it first writes a small header containing `blob`, the content length and a NUL separator. This canonical form makes the object type, size and exact body part of one identity.

Now we change only the final digit from seven to eight. The filename and byte length remain identical, but one content byte differs. When Git calculates the new object ID, will it replace the old blob or create another object?

PAUSE AND PREDICT

What happens after the one-byte content edit?

The old blob is overwritten

A new blob ID is created

Skip this check

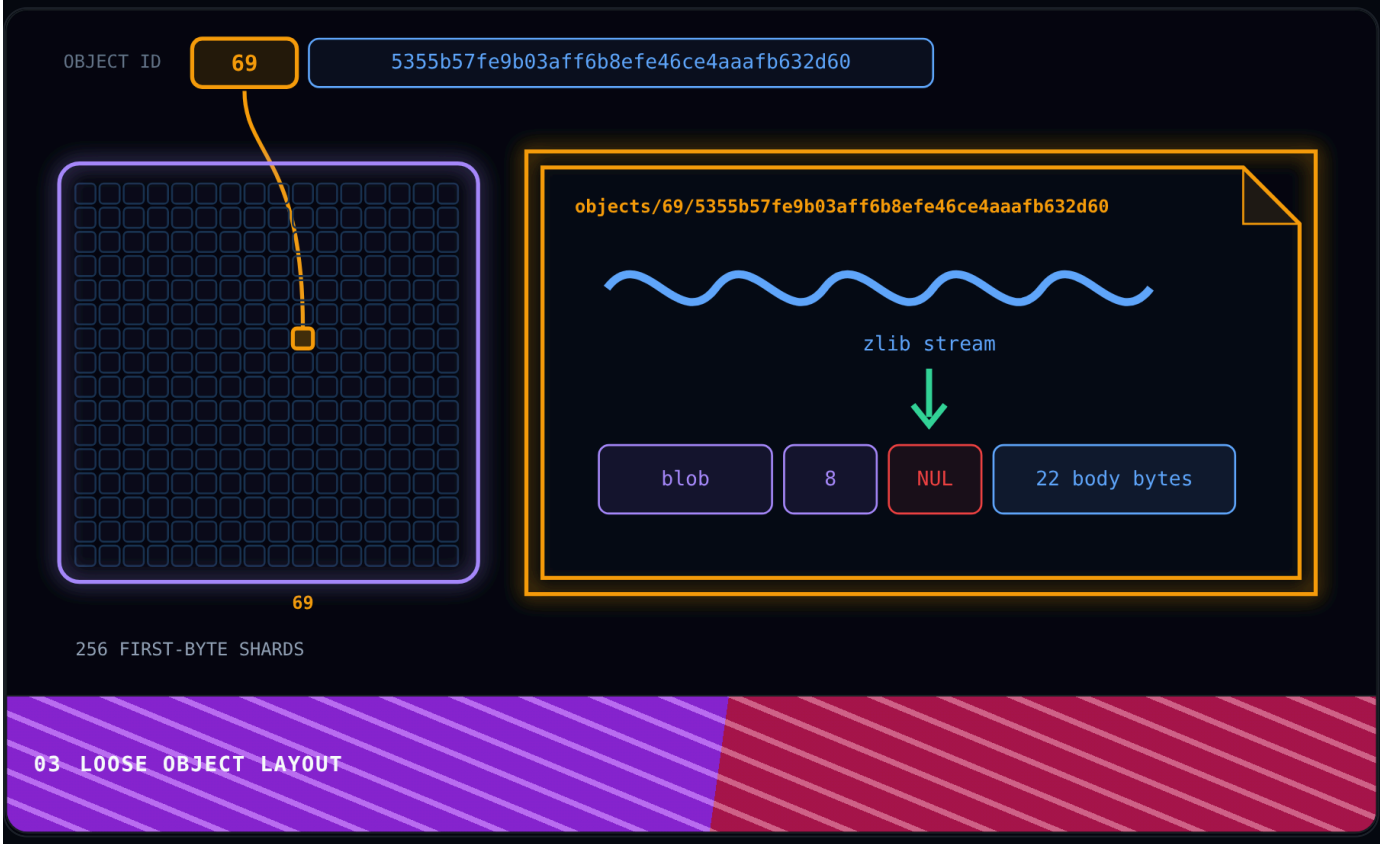
The one-byte edit produces a different digest, so Git stores a second immutable blob. The original still has its old ID and remains available to earlier snapshots. This is why an edit creates a new object instead of mutating history.

Switch the Content variant through both choices. Matching content reuses the existing object ID, while the one-byte edit produces a new immutable blob.

We now have the first rule of Git’s data model. Identical canonical bytes always lead to the same object ID, while any changed byte creates a different identity. Next, we will use that ID to locate the stored object on disk.

STAGE 3 OF 8

Loose Object Layout



The blob now has an object ID, so the natural next concern is where its bytes live. For a loose object, Git splits the first two hexadecimal digits from the rest. Those two pieces become a directory name and a filename.

Two hexadecimal digits can represent every value from `00` through `ff`, which gives Git 256 possible first-level directories. This fanout spreads objects across many small directories instead of letting one directory grow without bound.

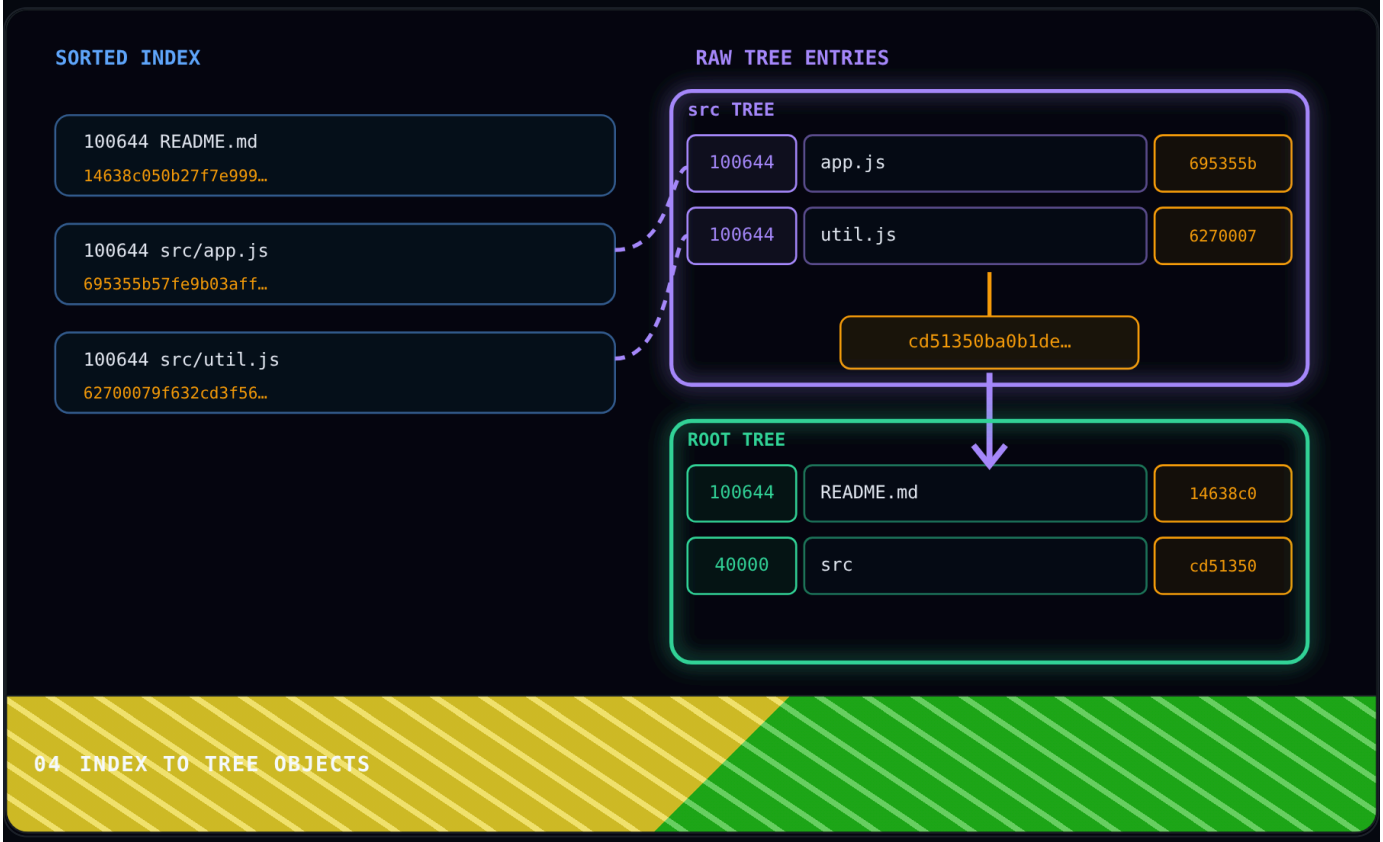
The remaining hexadecimal digits name the file inside that directory. Git compresses the canonical header and body with zlib before writing them there. Compression changes the stored bytes, but the object ID still describes the uncompressed canonical object.

Git can therefore calculate the full path directly from the object ID, without consulting a database or search index. When it inflates the file, it recovers the same header and body that originally produced that ID.

This stage separates logical identity from physical storage. The hash names canonical object bytes, while the filesystem path and zlib stream are only ways to locate and store them. Next, we will give these anonymous blobs filenames and folders.

STAGE 4 OF 8

Index to Tree Objects



04 INDEX TO TREE OBJECTS

Our blobs still do not know whether they came from `README.md` or `src/app.js`. Git's index supplies that missing structure. Think of it as a sorted staging record where each tracked path sits beside its file mode and blob ID.

Git first groups entries that share the `src/` folder. It writes each tree entry as a mode, a short name, a NUL separator and raw object-ID bytes. In this SHA-1 example, the raw ID uses twenty bytes rather than forty hexadecimal characters.

After Git writes the `src` tree, that entire folder has its own object ID. The root snapshot still needs to include `src`, so will it repeat every nested file or point to the completed subtree?

PAUSE AND PREDICT

What does the root tree store for src?

- **Both full nested paths**
- **One subtree object ID**

Skip this check

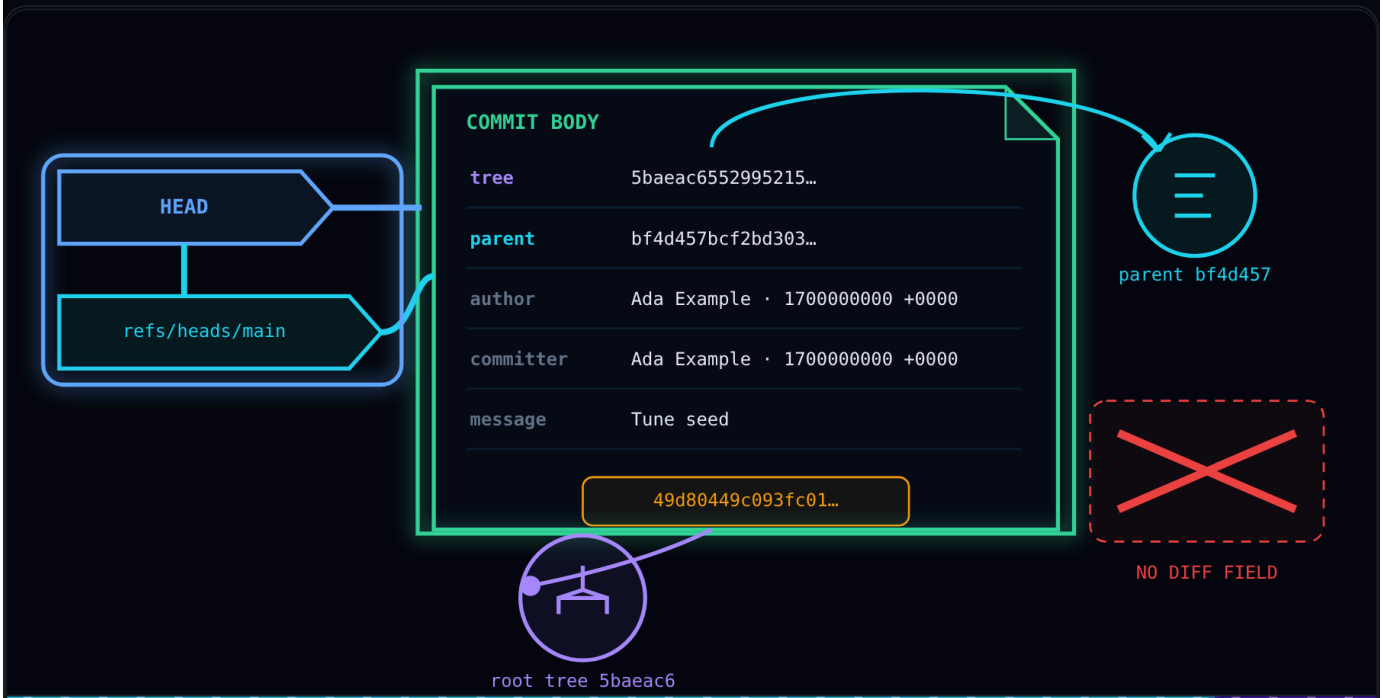
The root tree stores one entry named `src` that points to the completed subtree. It also stores the `README.md` entry directly. Because the child tree must have an ID first, Git builds directory snapshots from the deepest folders upward.

One root tree ID now commits to every tracked path, mode and blob beneath it. That is Git's complete directory snapshot, assembled from smaller reusable objects. Next, we need a history record that says

when this snapshot became meaningful.

STAGE 5 OF 8

Commit and Ref Resolution



05 COMMIT AND REF RESOLUTION

The root tree gives us a snapshot, but it says nothing about authorship or earlier history. A commit adds that context. Its body names the root tree, author, committer, time, message and any parent commits, then receives an object ID like every other object.

The tree pointer leads into the project snapshot, while each parent pointer leads backward to an earlier commit. The first commit has no parent and a merge can have several. Following these parent pointers is what turns separate snapshots into a history graph.

At this point, it is reasonable to picture a commit as a saved patch. We changed one file between the parent and current snapshots, so does the commit object store that textual diff inside its own body?

PAUSE AND PREDICT

Does a commit object contain the stored file diff?

- **Yes, the patch is embedded**
- **No, snapshots are compared later**

Skip this check

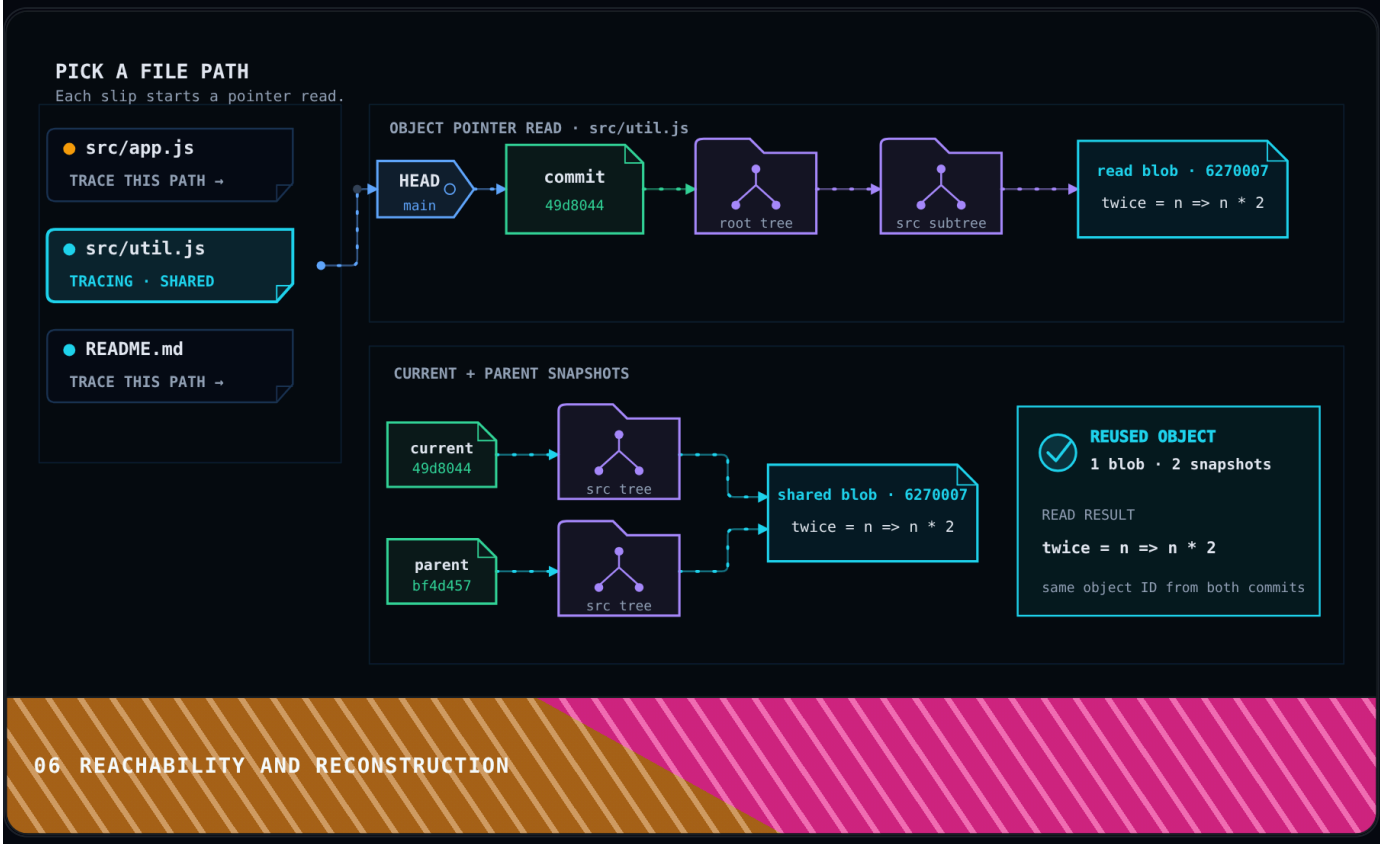
The commit stores the current root tree and parent IDs, not a patch. Git calculates a diff later by comparing two trees. Because raw commit IDs are awkward for people, the branch ref stores the current commit ID and `HEAD` usually names that branch.

This gives Git two kinds of identity. Commits, trees and blobs are immutable objects, while refs are movable names that can advance to newer commits. Next, we will begin at `HEAD` and follow these

pointers to recover a file.

STAGE 6 OF 8

Reachability and Reconstruction

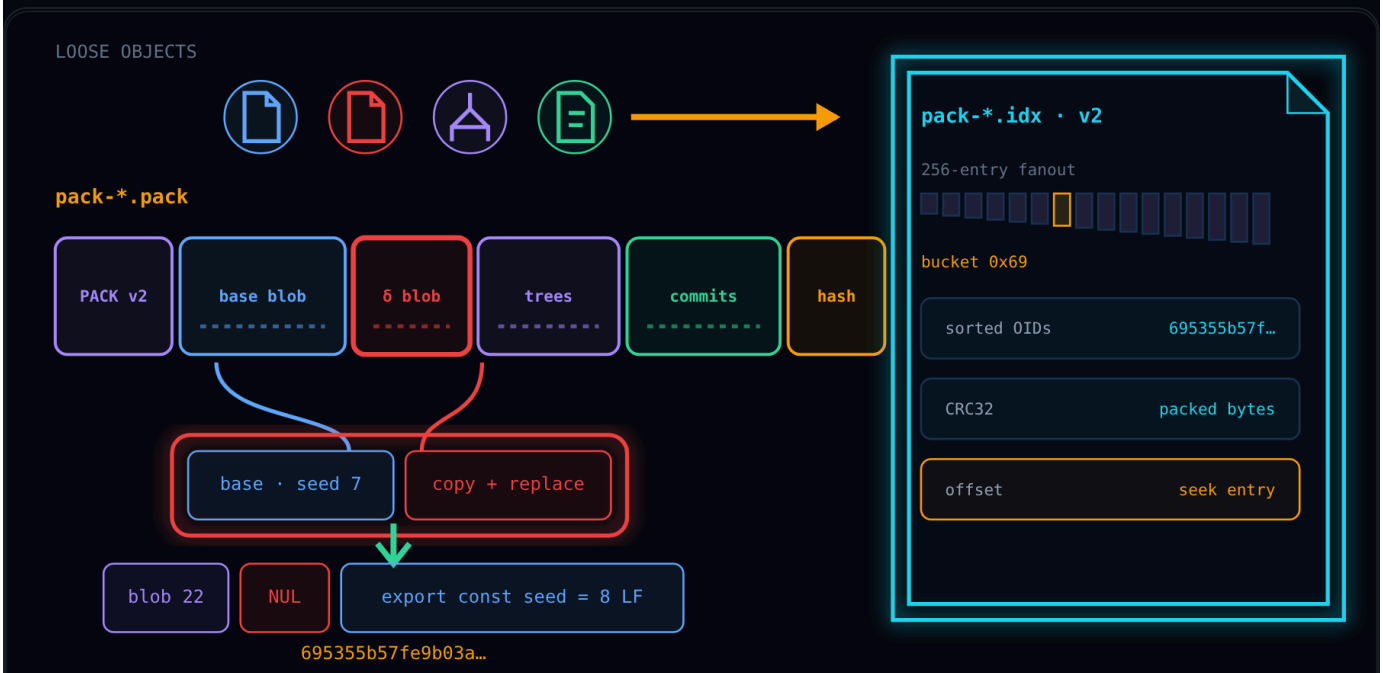


06 REACHABILITY AND RECONSTRUCTION

Now use those pointers yourself. Select a file, then follow HEAD to the branch, commit, trees and blob. The changed app reaches different blob IDs in two snapshots, while unchanged files converge on the same object. That convergence is Git’s automatic storage reuse.

STAGE 7 OF 8

Packfile and Index



07 PACKFILE AND INDEX

Loose storage is easy to understand, but a large repository would create an enormous number of tiny files. Git can combine many objects into one sequential packfile. The pack records its version, object count, compressed entries and a checksum for the whole file.

Combining objects creates a new lookup problem because Git should not scan the pack from the beginning each time. A companion index solves that problem. It uses a 256-entry fanout table, sorted object IDs, CRC values and byte offsets to jump near the requested entry.

Our two versions of `app.js` differ by only one byte, so storing both complete bodies may waste space. The packer can store one as instructions relative to the other, but does that delta representation change the logical object ID?

PAUSE AND PREDICT

Does deltification change the Git object ID?

- **Yes, the delta bytes are hashed**
- **No, canonical content is hashed**

Skip this check

The index locates the delta entry, then Git loads its base and applies the reconstruction instructions. The result is the edited blob’s original canonical content. Git verifies that reconstructed content against the same object ID it had before packing.

Packing changes how Git stores and retrieves objects, but not what those objects mean. Compression, delta choices and entry order can all change while canonical identity remains stable. Now let us step back and connect the entire data model.

STAGE 8 OF 8

The Complete Git Data Model

08 THE COMPLETE GIT DATA MODEL

We began with one ordinary file. Git combined its type, length and exact content into canonical bytes, then hashed those bytes. That is why identical content can reuse an object and a one-byte edit creates another.

Next, we separated identity from storage. The object ID calculated its loose-object path, while zlib compressed the canonical bytes inside that file. Neither the path split nor compression changed what the object meant.

The index then restored filenames and folders to our anonymous blobs. Git built child trees before parent trees, until one root tree ID represented every tracked path and file mode in the snapshot.

A commit added authorship, time, a message and parent pointers around that root tree. It stored a snapshot rather than a patch. Movable refs then gave people a stable way to find changing history tips.

Starting from `HEAD`, we followed explicit pointers through a branch, commit and trees to the requested blob. Changed paths reached new objects, while unchanged paths naturally reused the same object across snapshots.

Finally, Git reorganized many loose objects into a packfile. The companion index kept lookup direct and deltas reduced repeated bytes. Reconstructing canonical content preserved every original object ID.

The complete model follows one simple pattern. Immutable objects preserve exact content and history, pointers connect those objects into snapshots and refs choose the tip we care about. Storage formats may change underneath without changing the graph’s meaning.