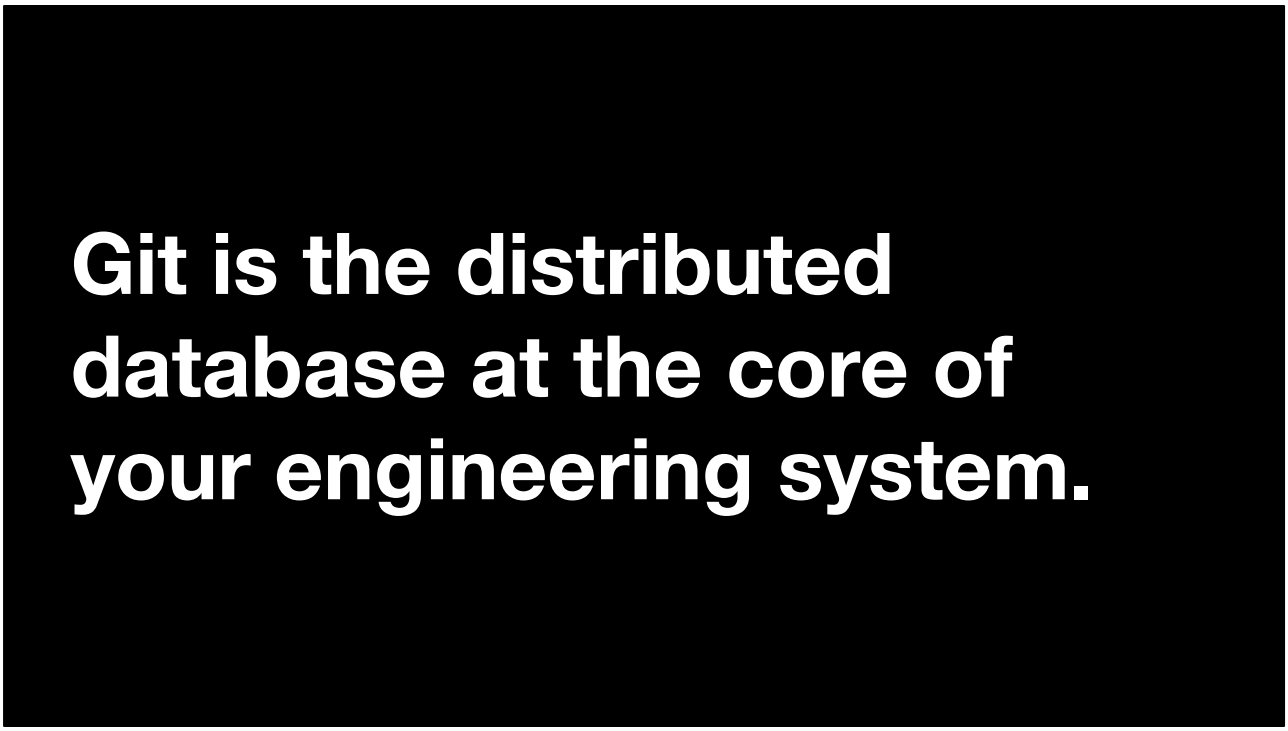
Hello, my fellow Git nerds!

I'm Derrick Stolee, an engineer at GitHub and a Git contributor.

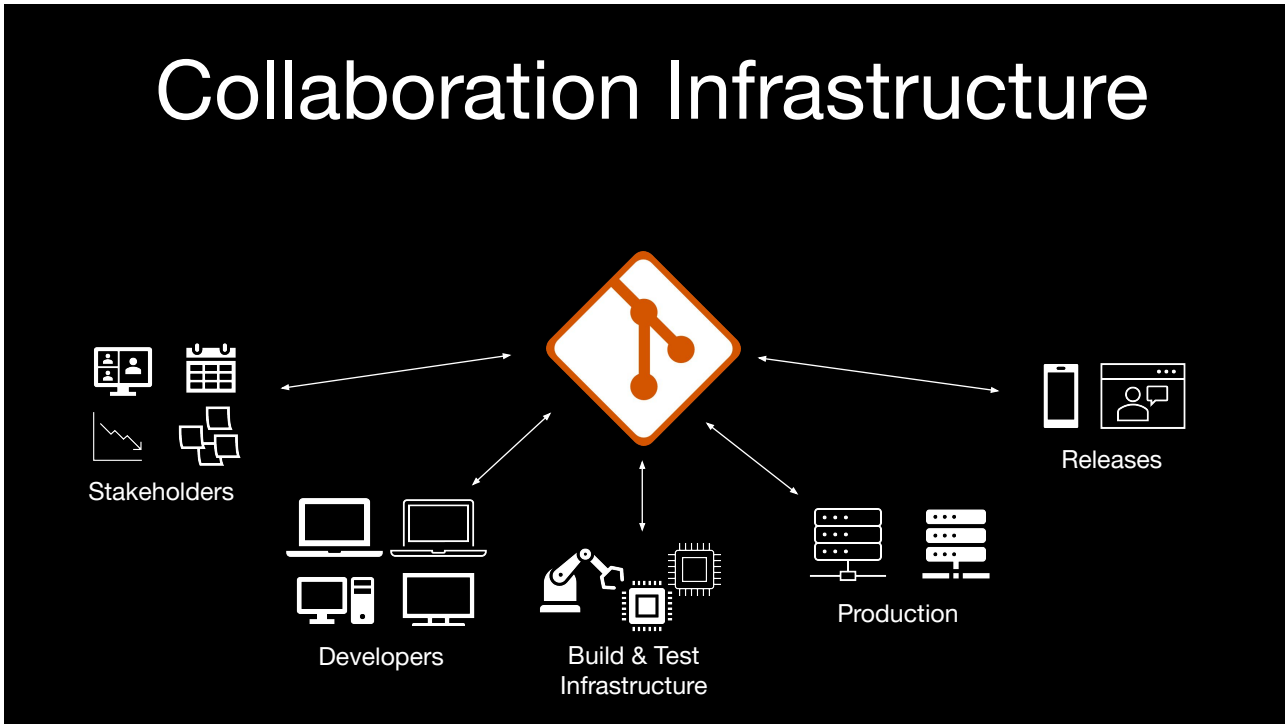
Today, my goal is to popularize an idea.

This idea should not be surprising or controversial to *this* audience, but I hope it gives you a framing device and a vocabulary as you leave this bubble of Git superfans and go back to your own organizations, spreading the good word.

Here is the main idea:



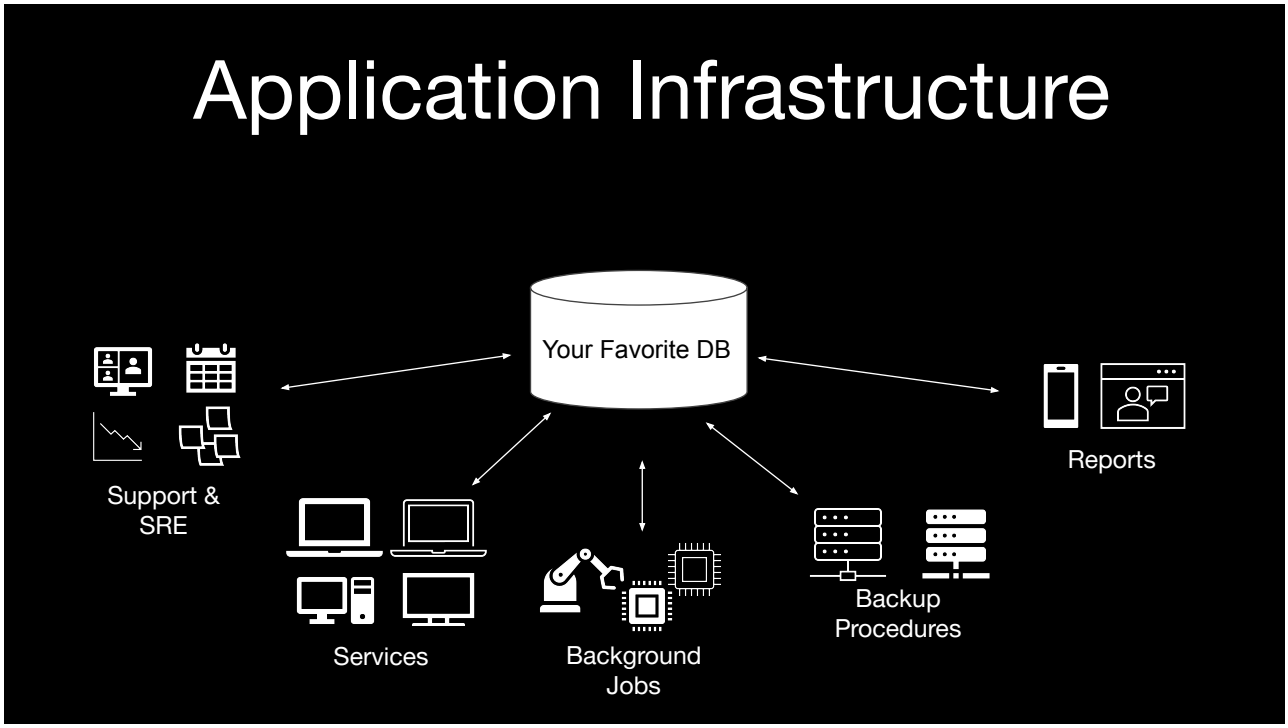
Git is the distributed database at the core of your engineering system.



When you think about it, Git is the center of your collaboration infrastructure.

Not only does Git allow multiple developers to do concurrent work on the same repository, but it also links with your build & test infrastructure, determines which versions you deploy to production or release to customers. Stakeholders in your organization may watch your repository to measure activity and progress.

All of these activities coordinate using Git as a communication medium.

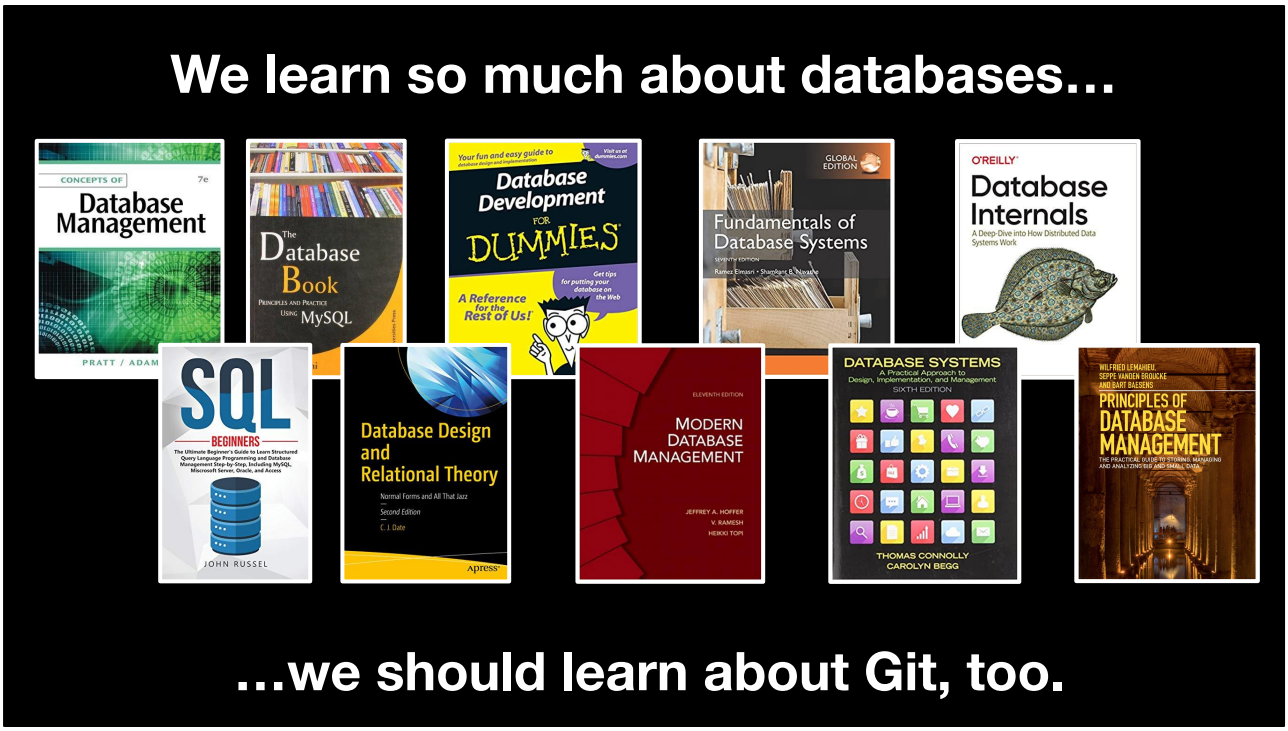


As a parallel, your application database is the core of your application infrastructure.

To persist application data, almost all applications use a third-party database application.

This database stores the information that is manipulated and served by application services. Background jobs process data async from user requests. Your infrastructure may have built custom backup procedures with failover remediation.

Let’s also keep in mind that the database health is monitored by support and SRE.



This leads to engineers learning a lot about how application databases work. There are thousands of books about how databases work and how you can write applications that take advantage of their strengths. No one bats an eye at the work required to understand your database and to interact with it properly.

Huge investments are made to build extra infrastructure around the database to make sure it remains healthy as your application grows.

(pause)

I'm here to say that everyone should learn about Git in the same way.

Database Concepts	Git Concepts
Storing (Table) Data	Git's Object Store
Queries	Git Commands
Query Indexes	Advanced Git Data Structures
Distributed Databases	Synchronizing Repositories
Sharding	Multi-Repo Organization

So here is my personal pairing of database concepts to Git concepts at the highest levels.

(click) At their core, databases store tables of data. Git stores objects in its object store.

(click) We manipulate and access database information using queries. Git's query language is its command line interface.

(click) Databases use specialized query indexes to speed up certain queries. Git has advanced data structures specifically for speeding up certain kinds of Git commands.

(click) Distributed databases have custom ways of remaining consistent and dealing with concurrent changes across database nodes. Git repositories communicate through fetches and pushes to synchronize after-the-fact based on user demand.

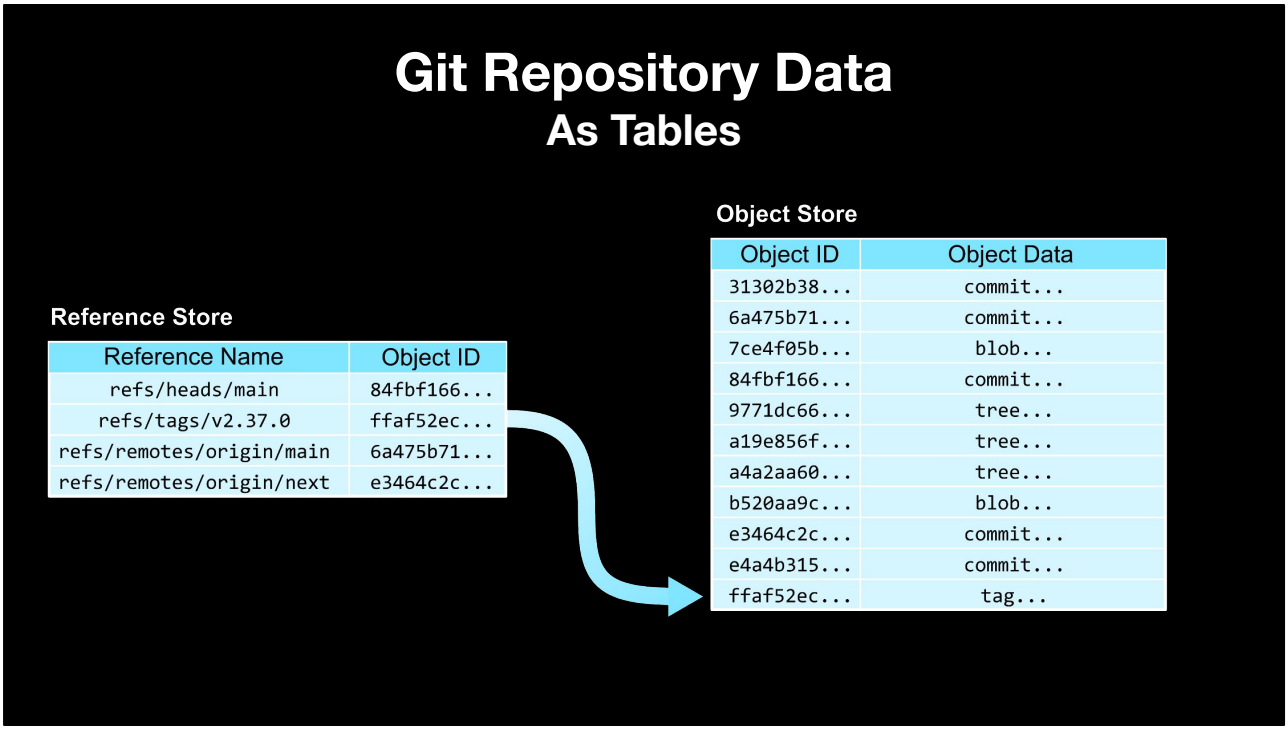
(click) Finally, when databases need to scale beyond the limits of a single node, application developers can use one of many sharding strategies. Git has similar strategies using multi-repo organization.

This is what I mean by equipping you with a framework and vocabulary. You can use these parallels when describing advanced Git features to your fellow developers, and hopefully starting from the common understanding of application databases will help bridge the gap to the Git concepts.

Today, I'll talk about my favorite parts of these concepts.

Git’s object store

Let’s start getting into some specifics, and we’ll start at the beginning, with Git’s object store.



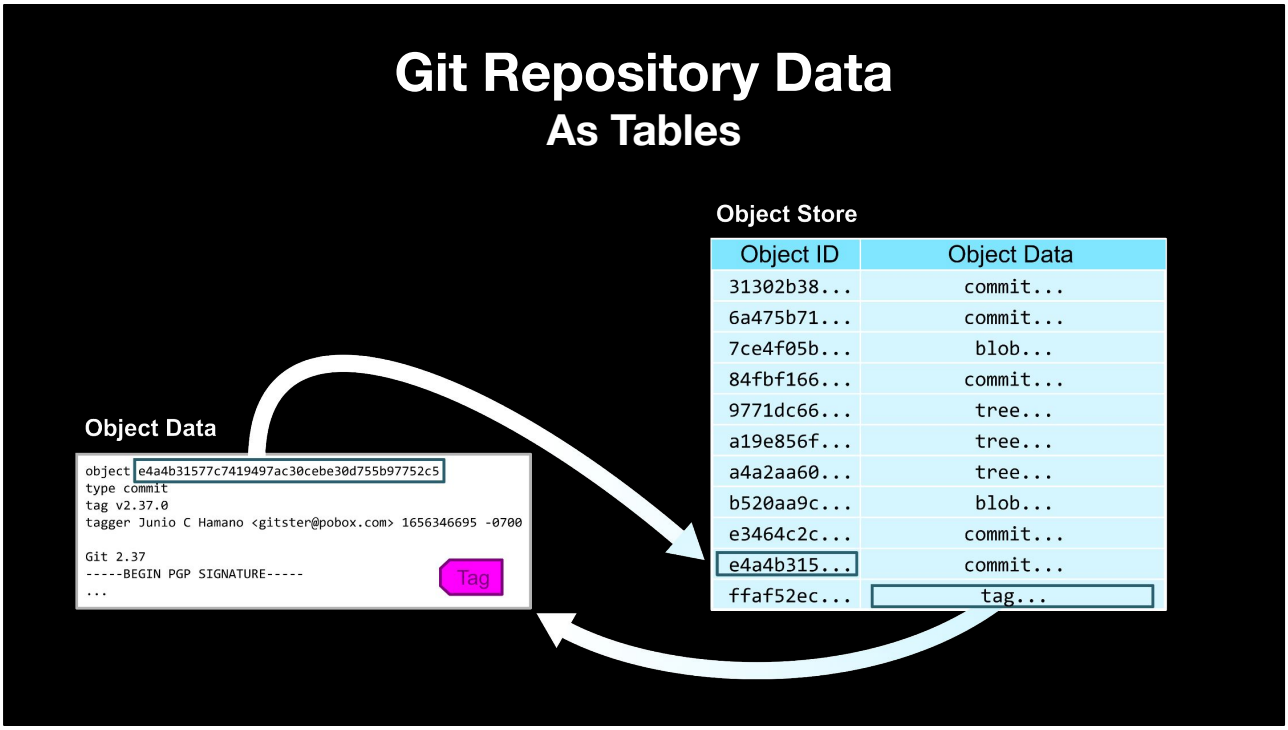
The main storage of Git repositories can be considered as two database tables:

The object store has two columns: an object ID and object data. For a given row, the object ID is the hash of the object contents column. This makes the object store a form of *content-addressable storage.*

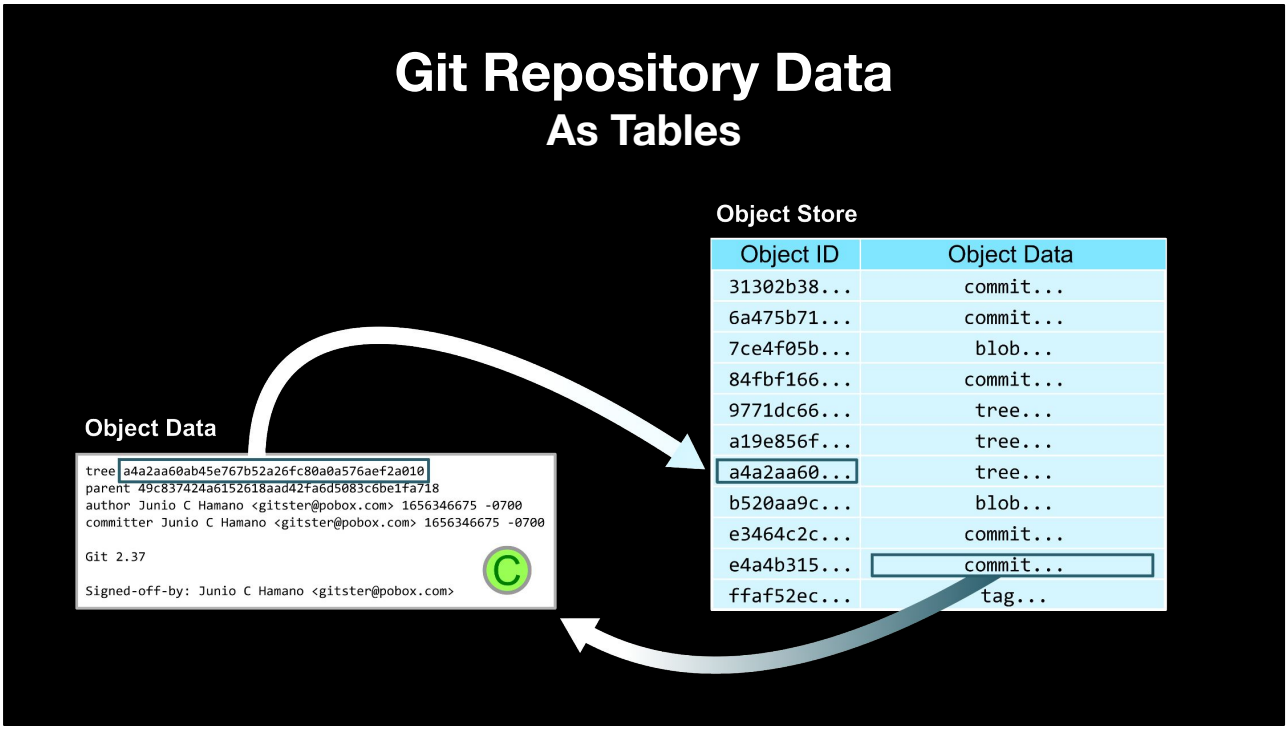
It's not too helpful to need the contents of what you are looking for before you find it, so Git has another table providing starting pointers.

The reference store table has two columns: a reference name and an object ID. The reference name is the primary key. These human-chosen names are pointers into the opaque object store, allowing us to gain a foothold into the contents.

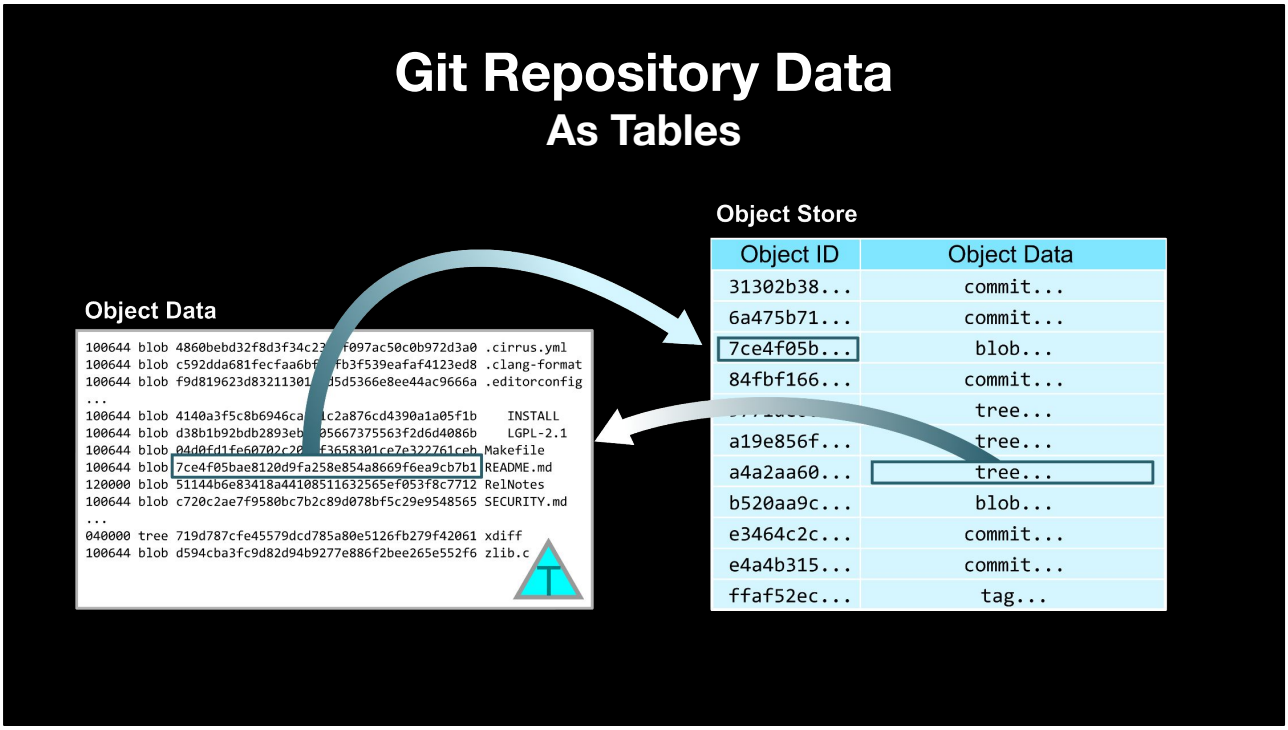
In this example, the "refs/tags/v2.37.0" reference points to an object in the object store.



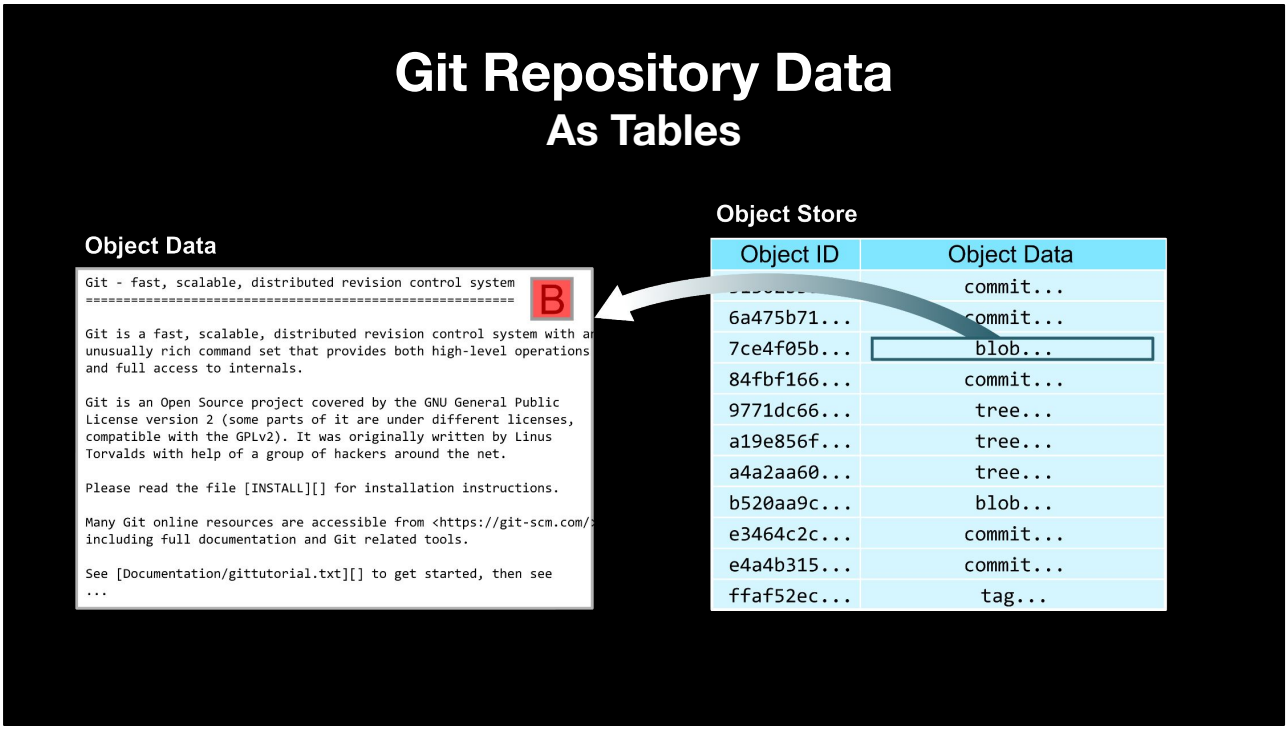
That object is an annotated tag, which points to a commit in the object store.



That commit specifies the object ID of its root tree.



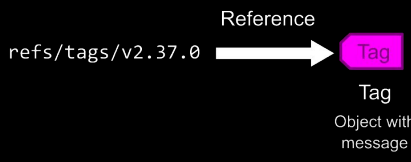
That tree contains an entry for the README.md file, whose object ID points to a blob.



That blob object stores that file's contents.

We've just taken several jumps through the object store just to find the README for Git 2.37.0.

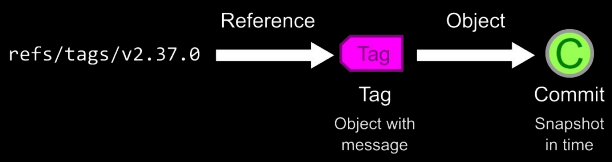
Git Repository Data
As Graph



We can review this lookup in a more abstract way as walking edges of a graph.

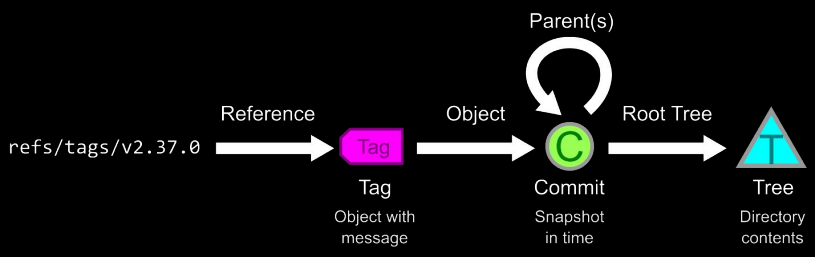
First, the reference points to a tag object.

Git Repository Data
As Graph



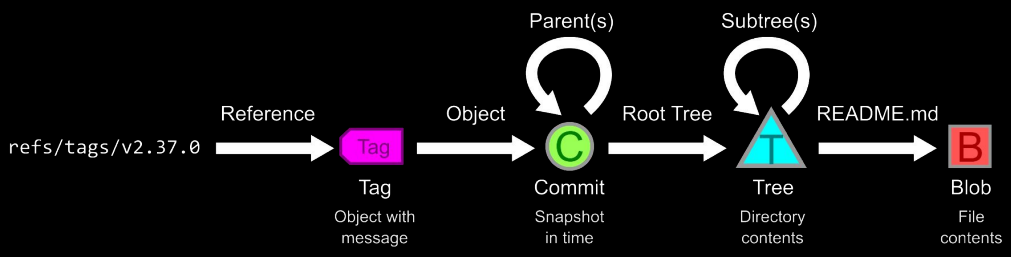
The tag object has an object pointer, pointing to a commit in this case.

Git Repository Data
As Graph



Commits have pointers to their parents and to their root tree.

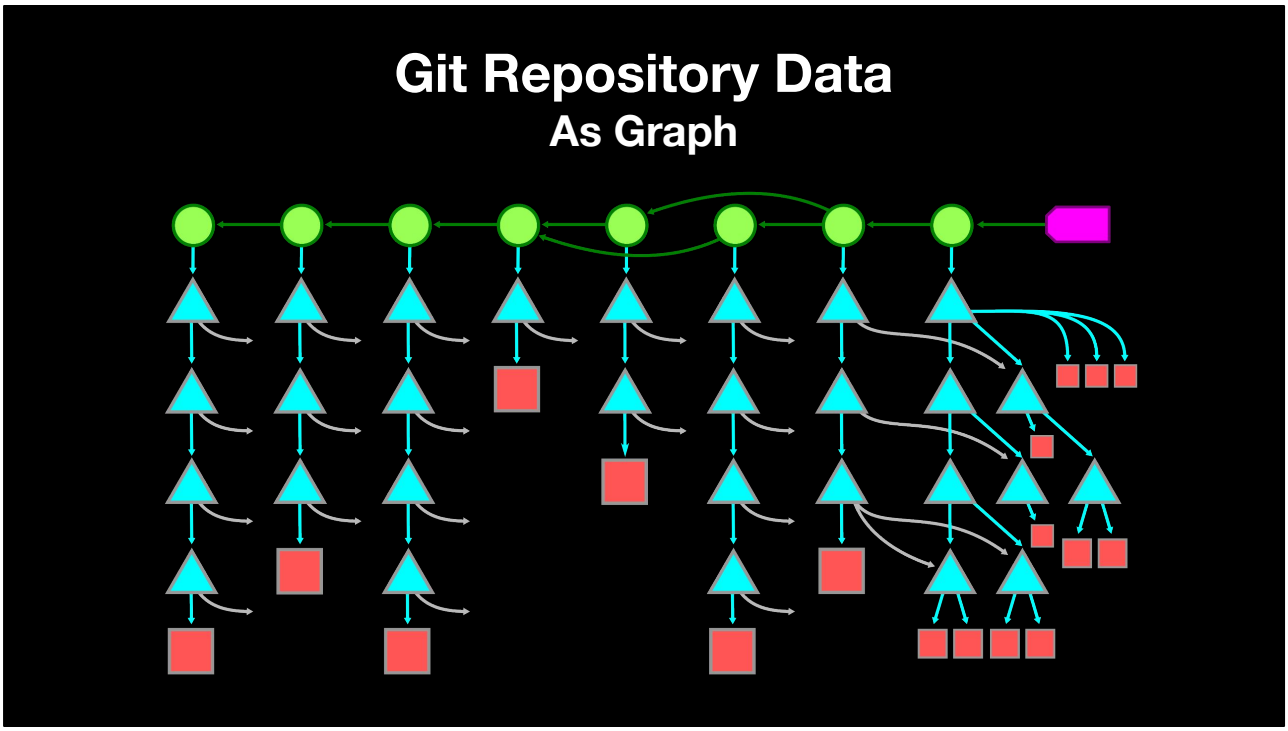
Git Repository Data
As Graph



Trees have entries pointing to other trees or blobs. We followed the entry for README.md to find the file contents for that file.

I'll use this representation for the rest of the talk, specifically:

- Circles are commits.
- Triangles are trees.
- And Boxes are blobs.



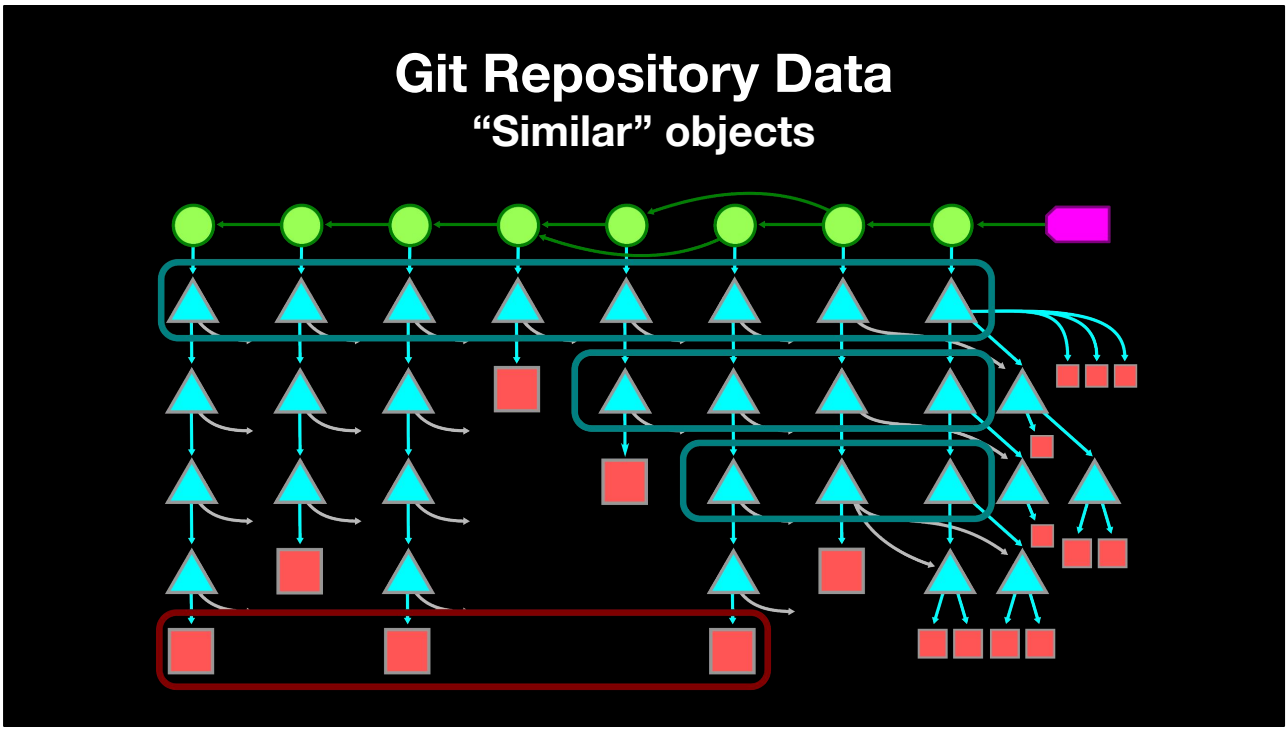
When seen in aggregate, the Git object graph can look like this.

I'll keep the commit history grouped at the top, with a row of root trees below.

As we go deeper into subtrees, we find that some tree entries are shared, so despite storing snapshots of worktree at each commit, many of the trees and blobs at two commits are actually shared.

This Merkle Tree representation is the first way that Git keeps its object store small as users make changes.

However, if we are storing full copies of every version of every file, then that would also grow too quickly for most use cases.



Here, I’ve grouped some objects that appear at the same path across multiple commits.

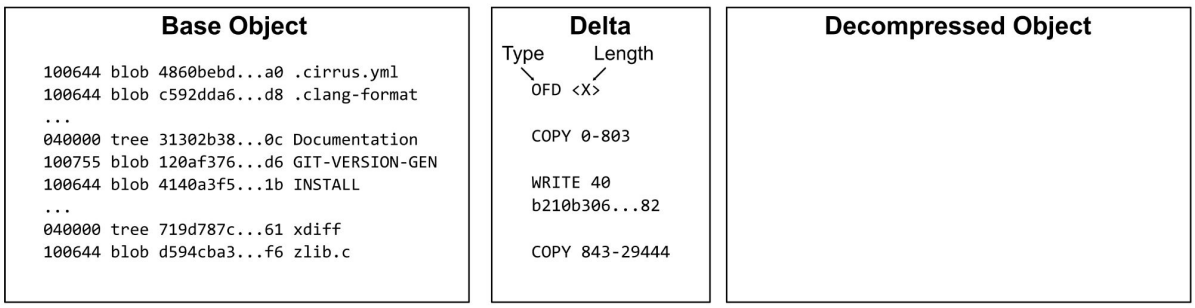
The top row of root trees all represent the base directory of the worktree.

These three blobs at the bottom might represent the same source code file.

One thing Git can expect is that these objects share significant portions in common, because as software developers we tend to modify a small portion of the repository at a time.

Git uses a particular kind of compression when objects have a lot of common data: delta compression.

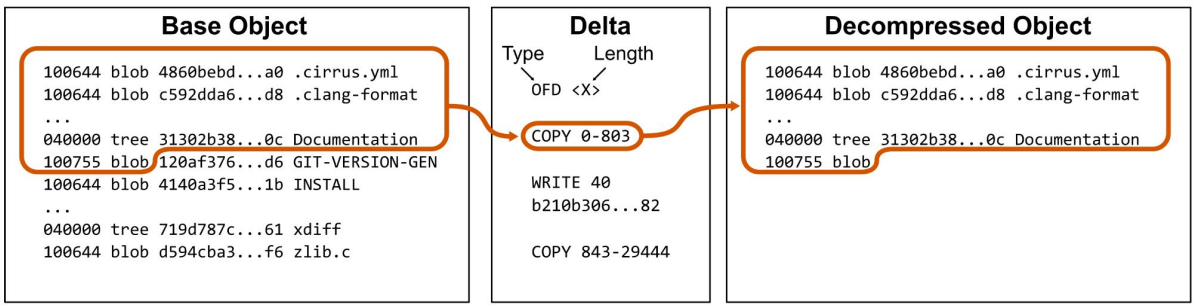
Object Storage
Delta Compression



Here, I'm showing and example tree object as the base object as well as an example delta object.

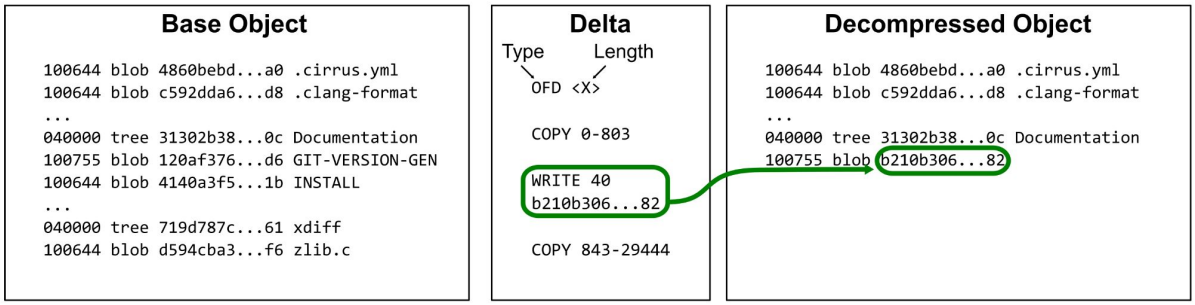
Deltas are instructions to help construct an object based on copying regions from a base object and writing new regions at certain points.

Object Storage
Delta Compression



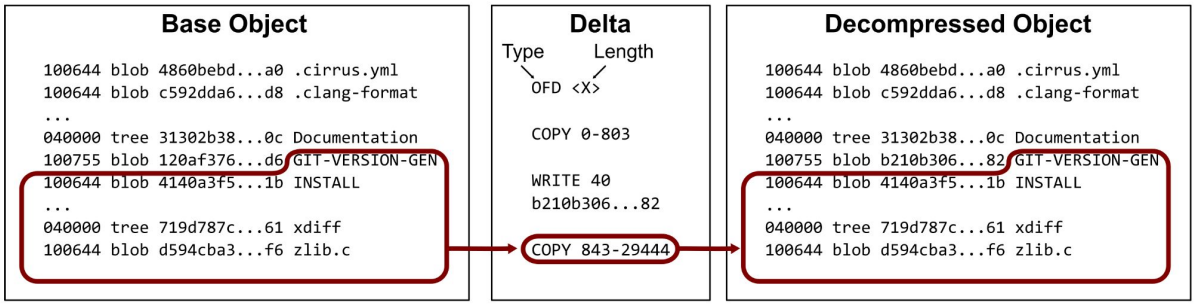
We start by copying the initial segment of the tree up until the object ID of the GIT-VERSION-GEN entry.

Object Storage
Delta Compression



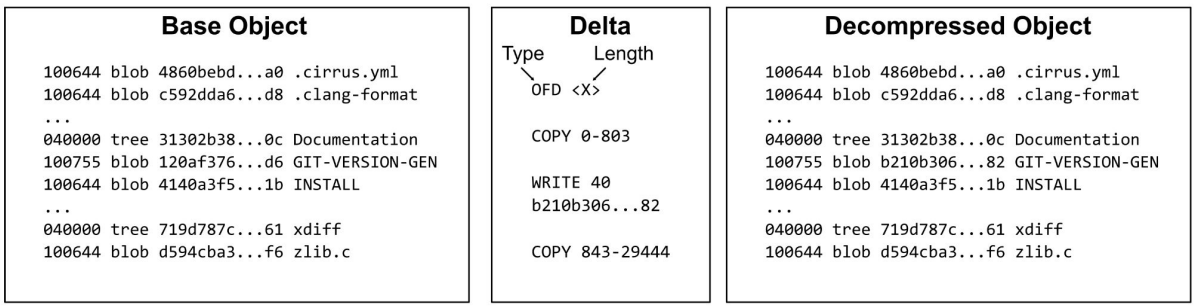
Then, we write a new section of data corresponding to a different object ID.

Object Storage
Delta Compression



Finally, we copy the remaining data from the original tree, starting with the filename for the GIT-VERSION-GEN entry.

Object Storage
Delta Compression



At the end of this process, we have constructed the decompressed object, but used significantly less data to store the two objects than if we did not use dela compression.

This type of compression works quite well for trees, but also works well for most blobs, assuming the blobs store plain-text files such as source code and documentation.

Object Storage
Pack Files

Git has a custom file format that can take advantage of delta compression: pack files.

Pack files store multiple objects by concatenating their contents into a single file. The objects are “packed” together. In addition to full object contents, deltas can also be stored, as long as their bases are also in the packfile.

This on-disk storage is not the only way git uses this pack-file format, though!

Sending objects between repositories
Pack Files

In fact, the format used to store compressed object data on disk is also used to share objects over the network between repositories.

Let’s explore how repositories synchronize efficiently.

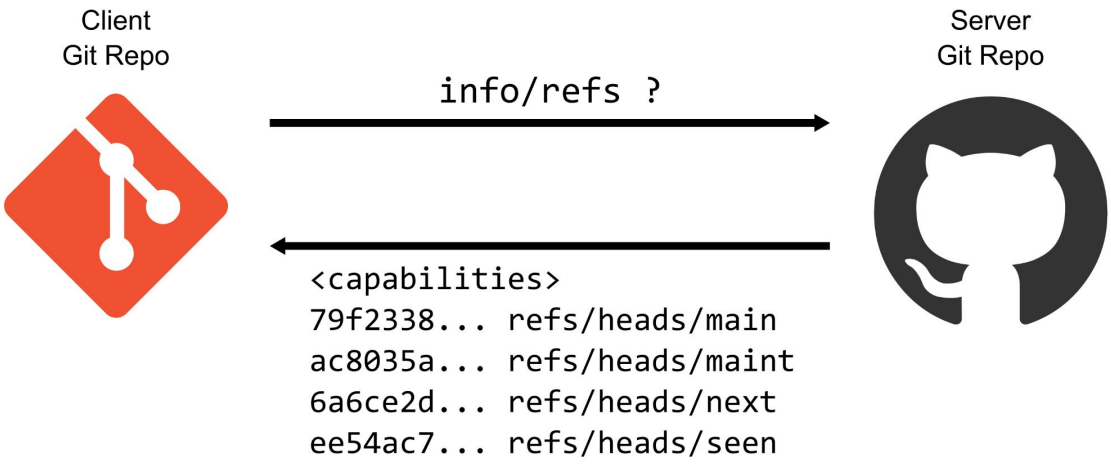
Repository Synchronization

Fetching new objects

The following process outlines the behavior between a git client and a git server during a “git fetch” command.

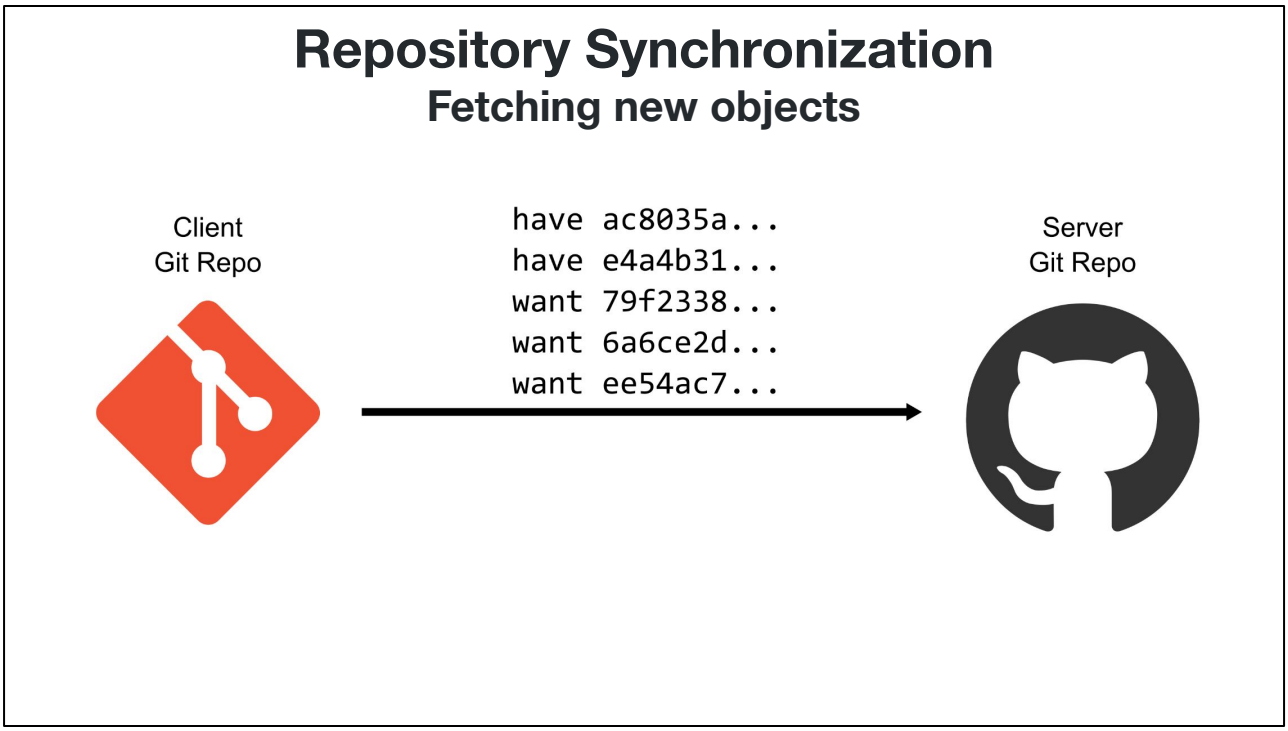
Repository Synchronization

Fetching new objects



First, the client asks the server for the ref advertisement, which is a list of references and their current object IDs from the server's perspective.

The client takes this information and decides which references are important as well as which object IDs are not present on the local machine. The rest of the communication is done via object ID, in case the remote server changes ref positions.



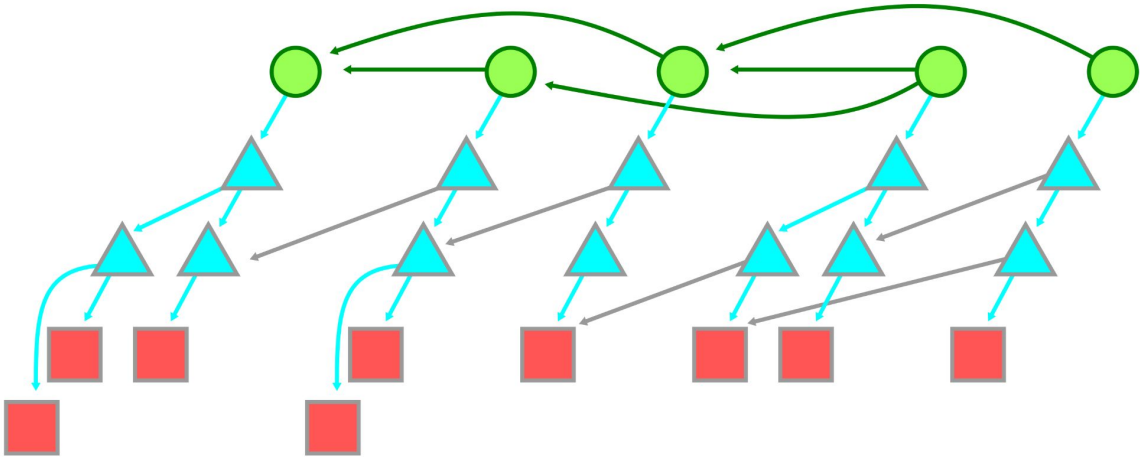
The client then sends a list of object IDs, each of which is marked as a “have” or a “want”.

The “want” IDs are objects that are not in the client repository, but are referenced by requested refs.

The “have” IDs are objects that are in the client repository, and the client guesses are on the server repository, based on previous records of the server’s references.

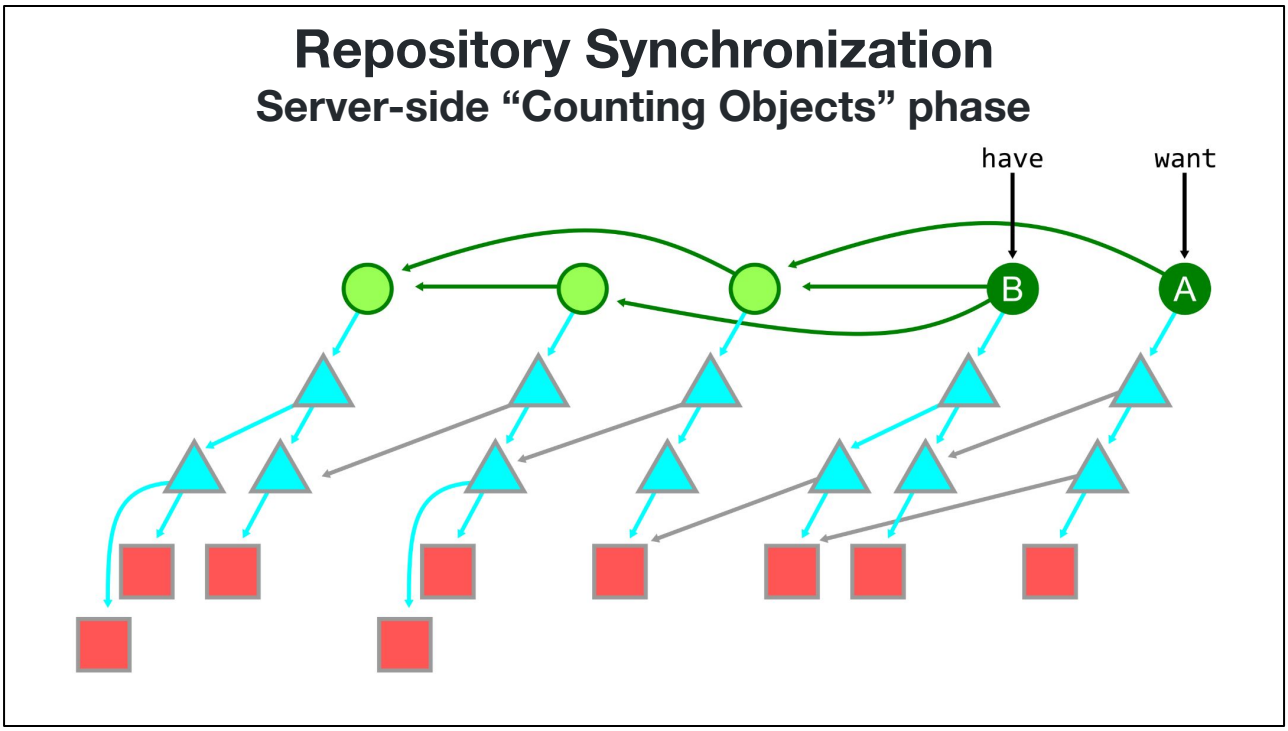
Repository Synchronization

Server-side “Counting Objects” phase

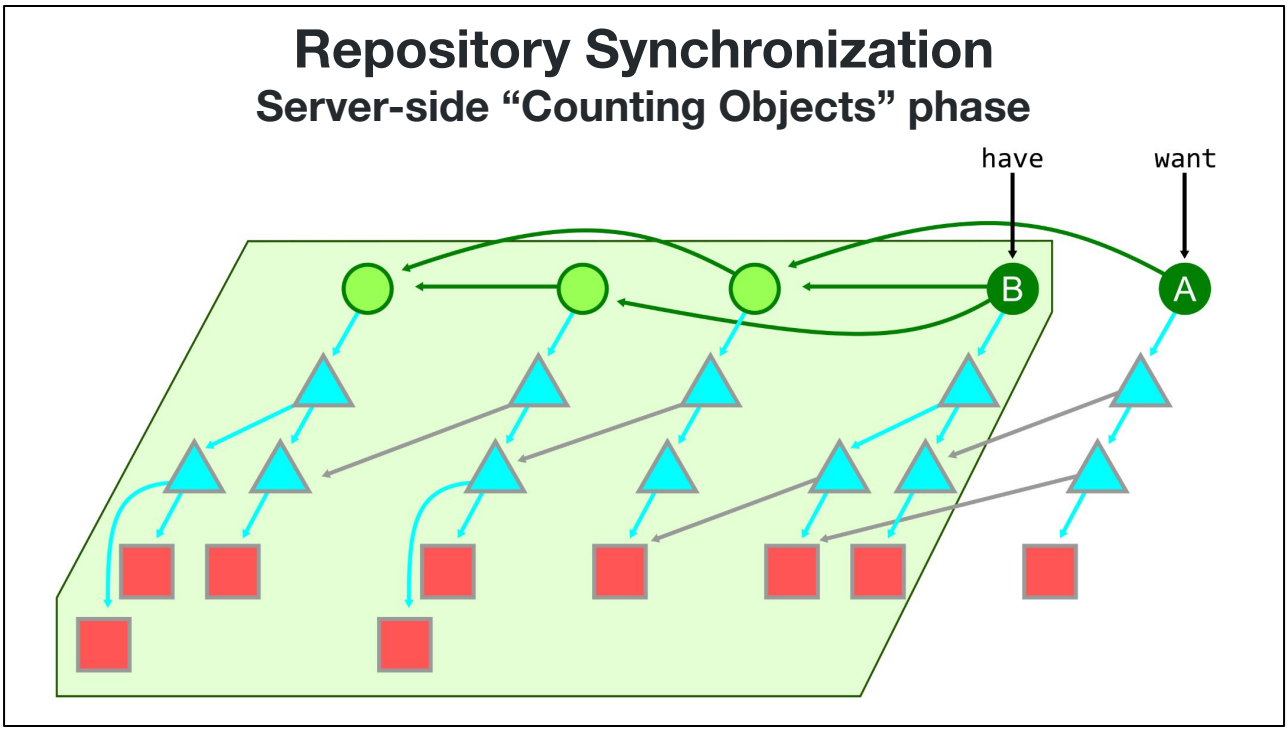


At this point, let’s take a look inside what the server is doing.

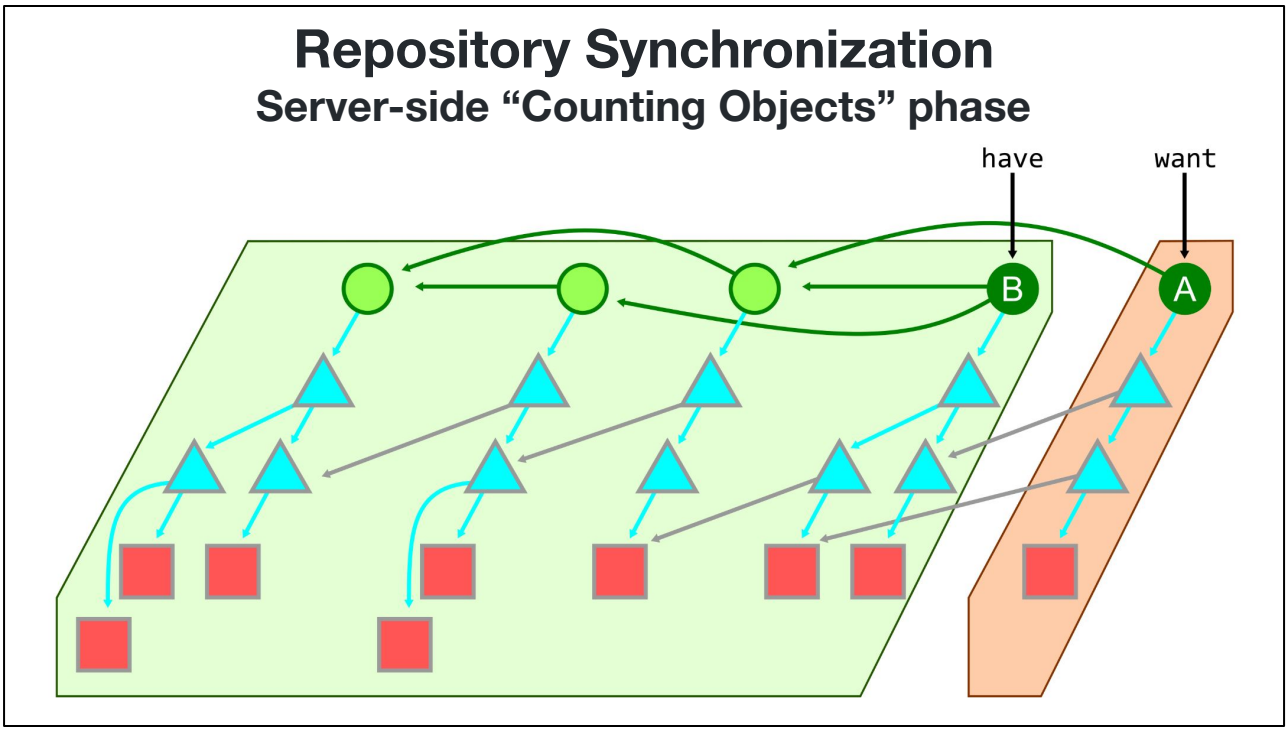
Here is an example object graph, including commits, trees, and blobs.



Suppose the client sends one have and one want. It wants the commit A and has the commit B.



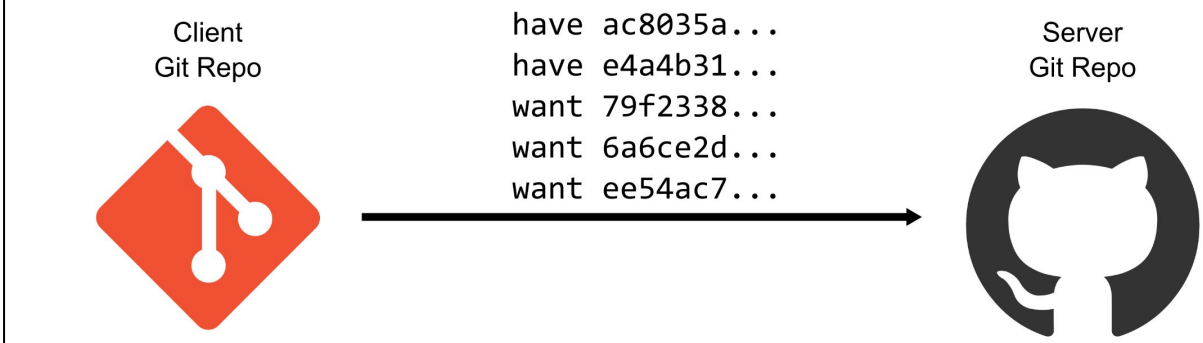
The server infers that the client has everything reachable from the haves, giving this region of objects.



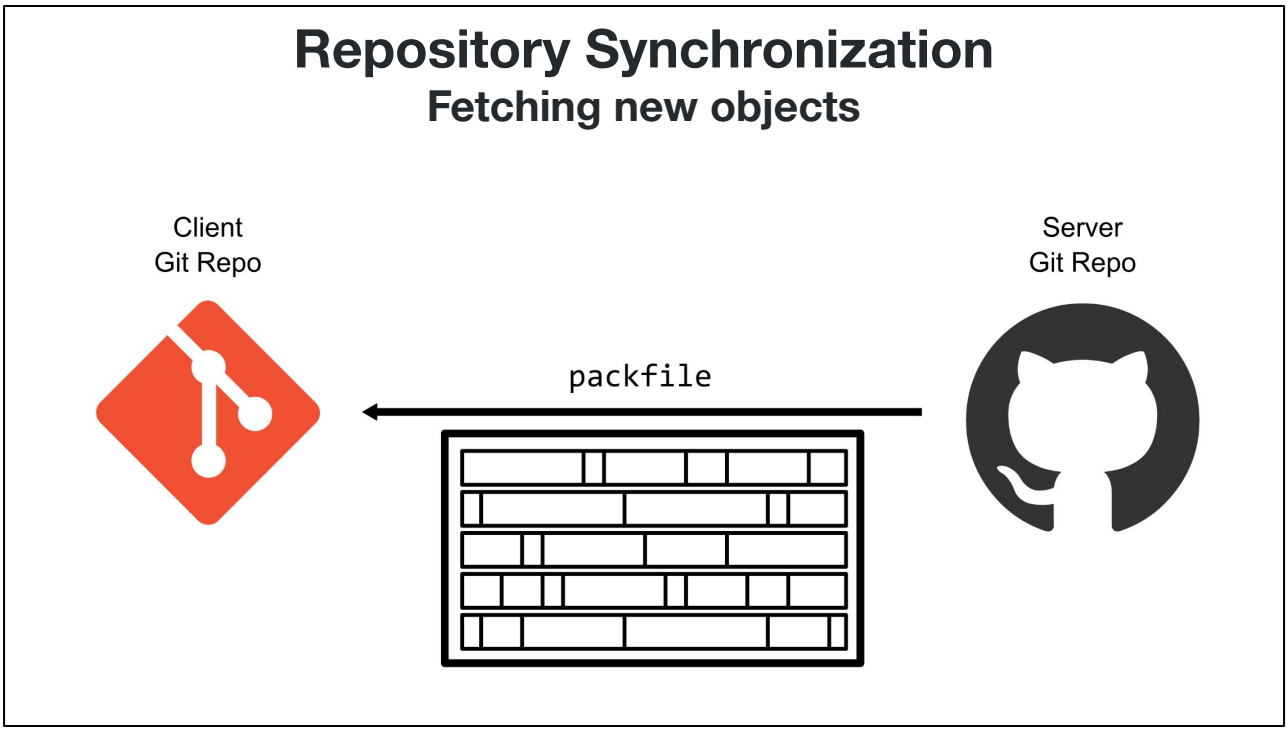
The server then determines which objects are reachable from the wants but _not_ reachable from the haves. These are the objects that the client needs.

Repository Synchronization

Fetching new objects



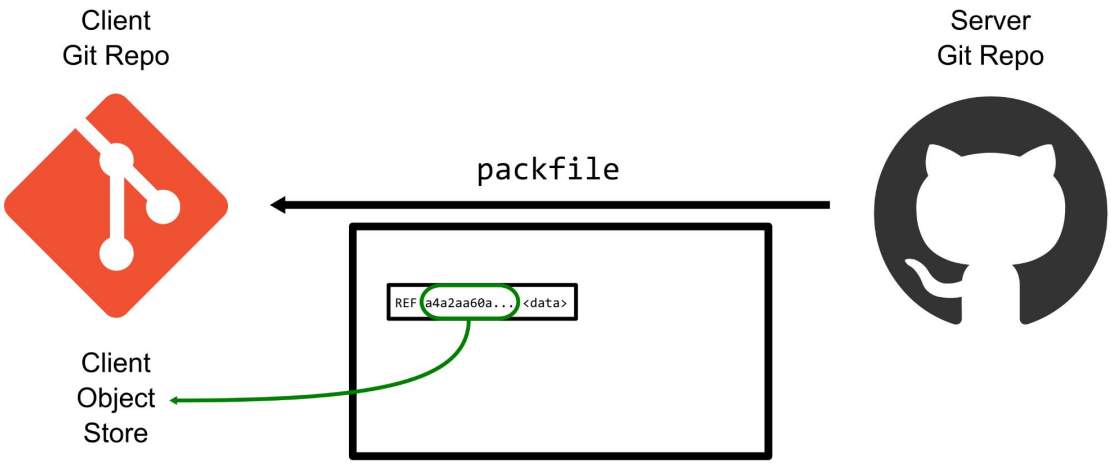
Back to the server/client interaction, the client may send a small number of haves and wants, but desires a possibly large set of objects.



The result is that the server sends a packfile filled with object data. This pack contains the objects from that object graph walk: objects reachable from wants but not reachable from haves.

This process allowed these two repositories to find a set of objects that the client needed without each side listing their full contents. This synchronization is specialized to use the object graph in creative ways.

Repository Synchronization
Fetching new objects



In addition to only providing the newer objects, the packfile sent over the network can also use delta compression.

These deltas can use offset deltas that point within the pack-file, but also “reference deltas” can point to a base object ID that is expected to be present on the client machine based on the list of “have” objects. This allows even further compression during synchronization.

What can *I* do about this?

You might be thinking: “It’s nice that Git has my back and is doing smart things under the hood, but what can *I* do about this?”

You are in control of your repository.

You determine its shape.

You can influence the norms of your organization.

Quick Git Tips

Run `git maintenance start`
Fetches in the background
Repacks incrementally

Use good repository hygiene
No large binaries
Don’t commit build outputs

I have two quick tips to share with you before going into some bigger picture items.

The first is that you should run “git maintenance start” in your favorite repositories. This will start fetching from your remotes on an hourly basis, reducing the time spent synchronizing in your foreground “git fetch” operations. It also repacks your object store incrementally on a nightly basis to keep things running smoothly and reducing the disk space required for your repository.

The second is that you should practice good repository hygiene. You want to take advantage of Git’s delta compression whenever possible. The good news is that files that do not delta compress well also do not tend to diff well or merge well, so they do not present as useful changes in your pull requests. To fix this, you should remove large binaries from your repositories, especially those that change often. Also, you should avoid committing files that are created by your build process. Only store build *inputs* in your repository, not *outputs.*

The remaining ideas I have to share are about macro-scale repository organization. They come from the concepts of sharding databases when application databases grow larger than what a single node can support.

Un-Sharded Repository
Monorepo

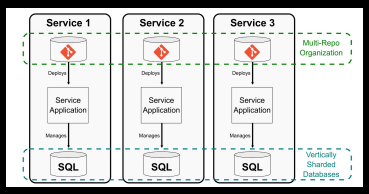
Before talking about sharding strategies, I should first mention that the lack of sharding is the basis of the monorepo organization.

Monorepos are a great idea, if you're careful and using good repo hygiene. As your repository grows, it becomes more important to use the advanced Git features that allow focusing on a small subset of the repository, like partial clone and sparse-checkout.

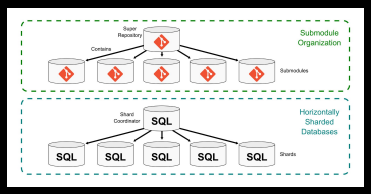
I've talked about monorepo scale a lot in the past, and it's been covered quite a bit today, so let's focus instead on these sharding strategies.

Scaling Repositories

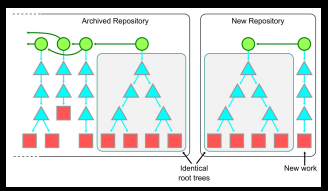
Vertical Sharding (Multi-Repo)



Horizontal Sharding (Submodules)



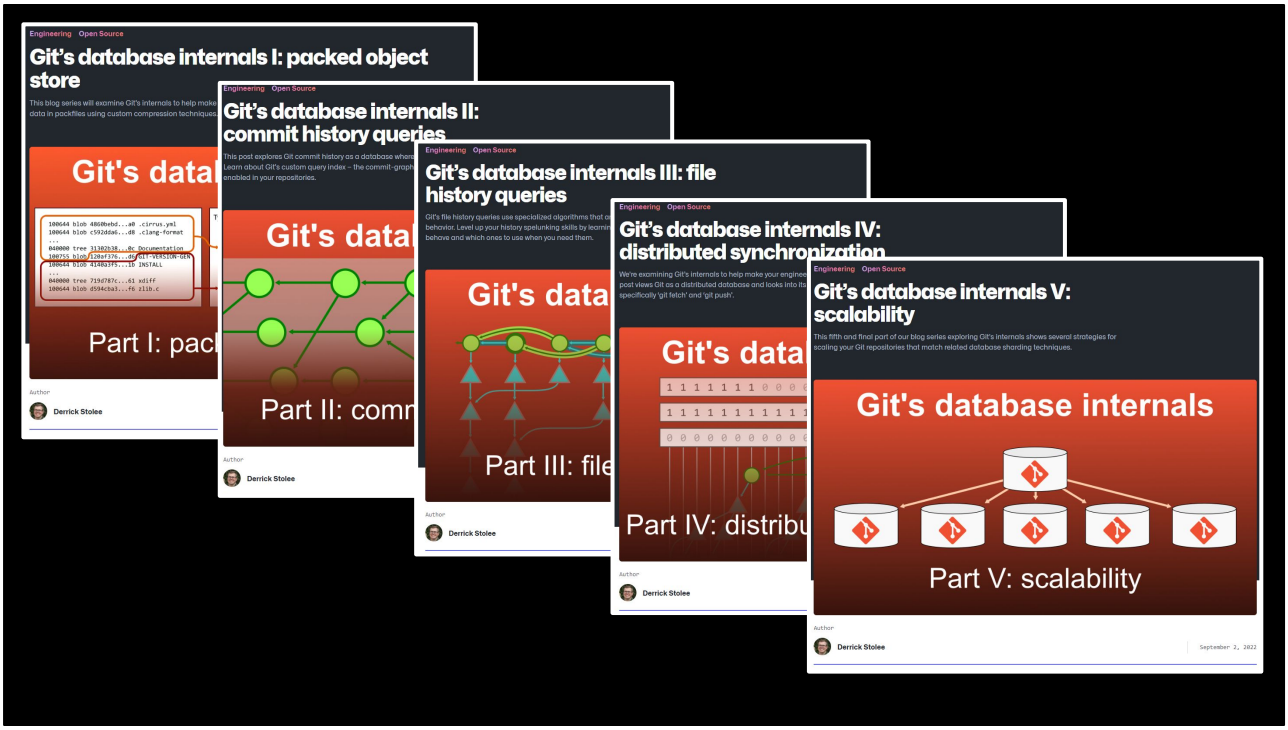
Time-Based Sharding



There are different ways to shard your repositories, depending on which works best for your situation.

Each of these strategies take inspiration from database sharding strategies, such as vertical sharding, horizontal sharding, and sharding of time-series databases.

Unfortunately, I don't have time to go deep into these strategies today.



I wrote a five-part blog series on the GitHub Engineering blog that goes super-deep on all of these concepts. Hopefully, this talk inspires you to dive deeper into these ideas, or to use them as reference in the future.

Git is the distributed database at the core of your engineering system.

And if you remember nothing else of what I've said, let me repeat the core concept that you should remember.

Git is the distributed database at the core of your engineering system!

Thank you!