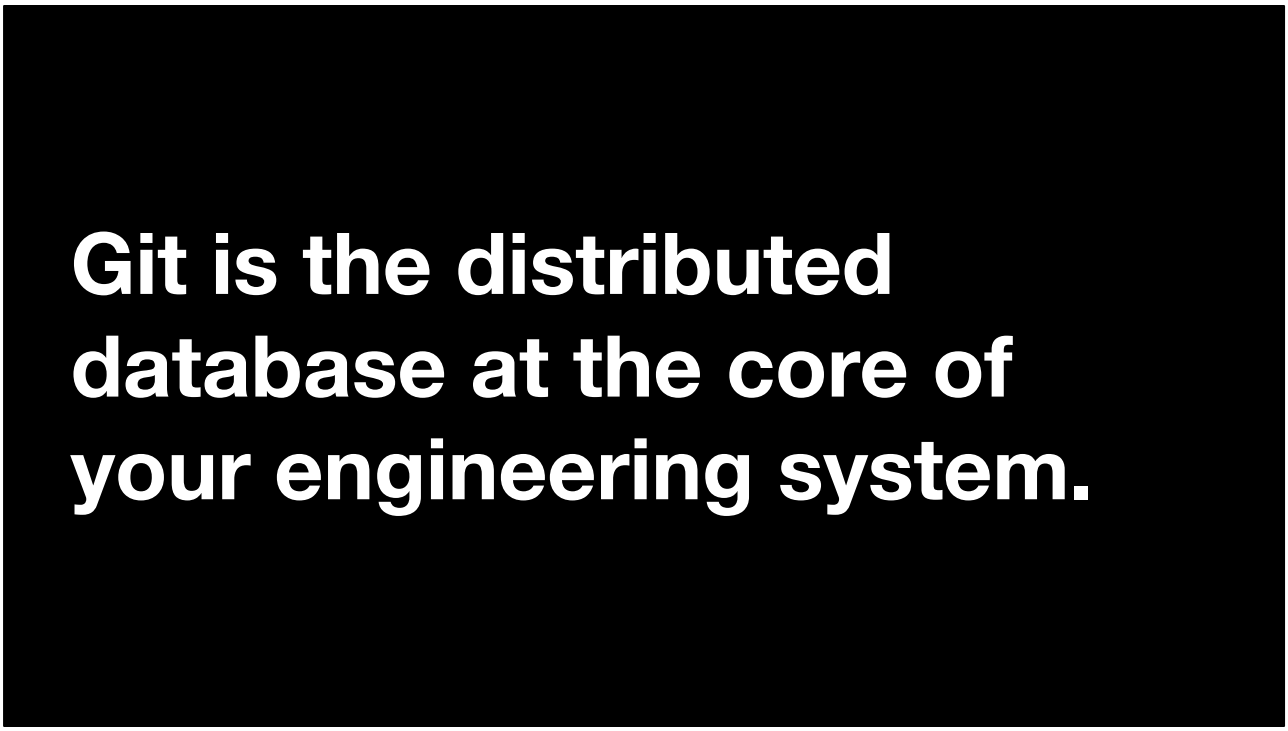
Hello, my fellow Git nerds!

I'm so excited to be surrounded by like-minded folks such as you.

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.

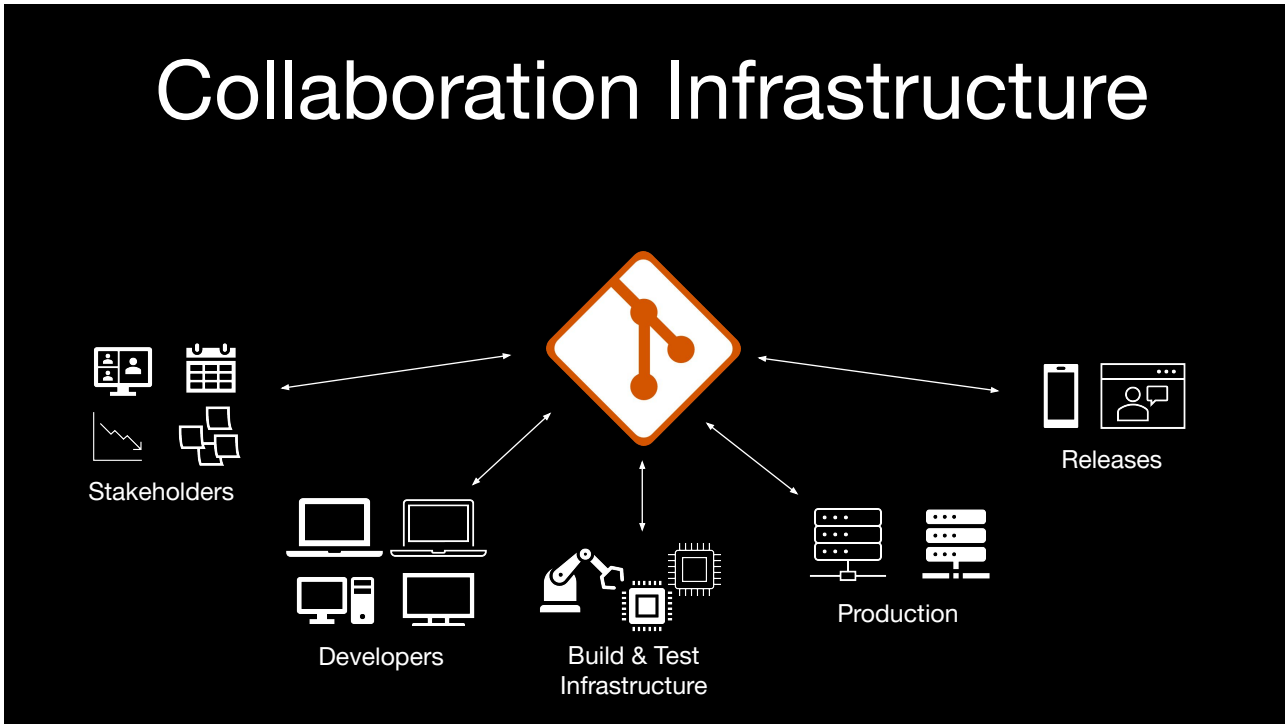
Here is the main idea:



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

(pause)

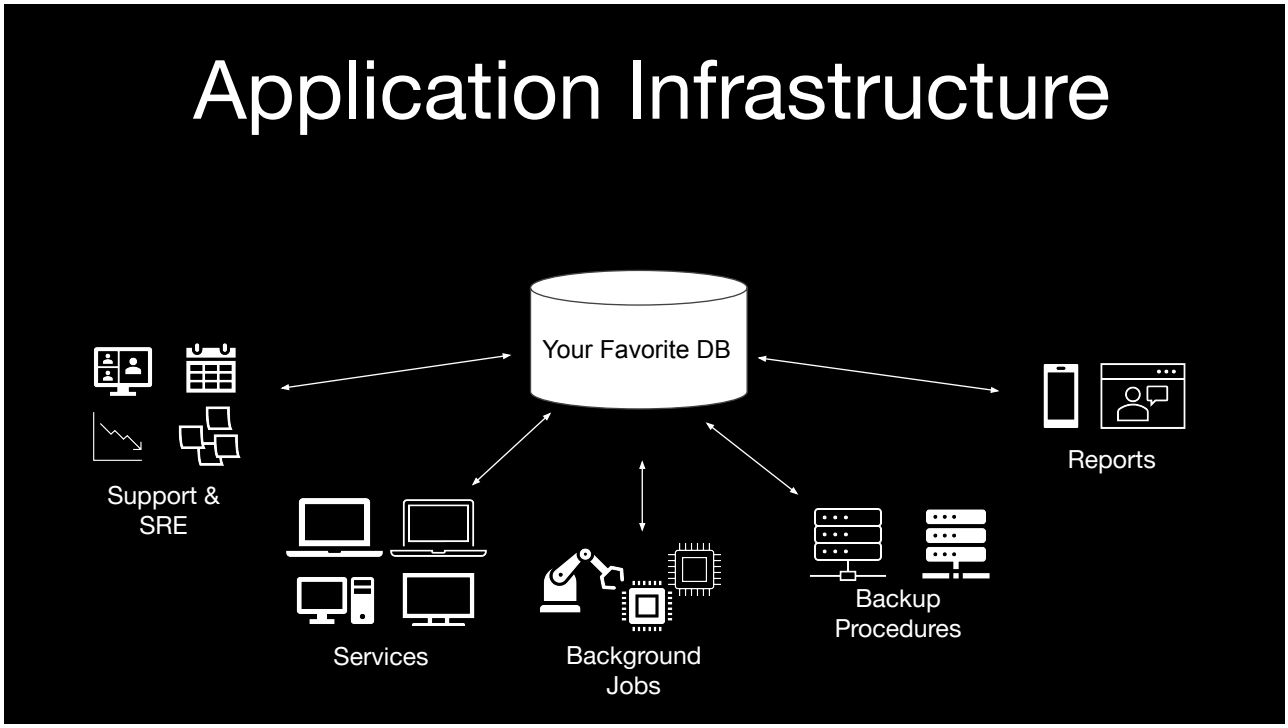
Git is the distributed database at the core of your engineering system.



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.

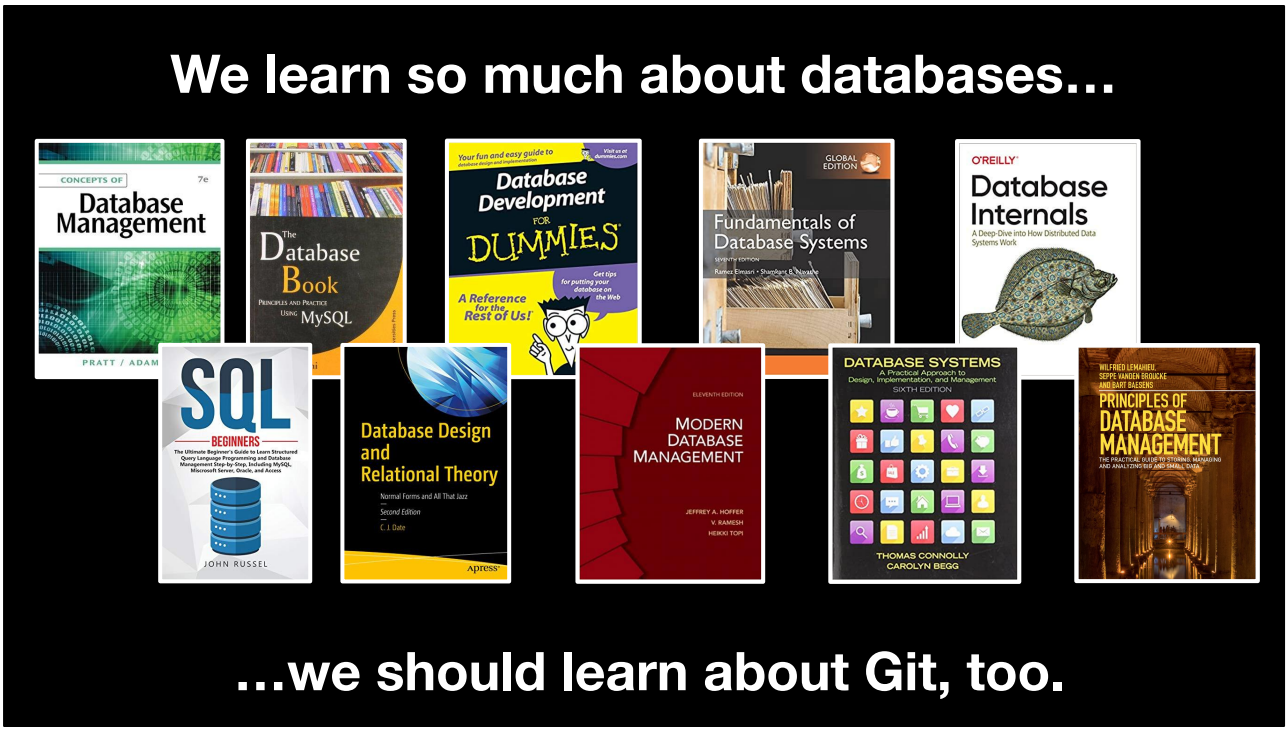


As a parallel, your application database is the core of your application infrastructure.

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.

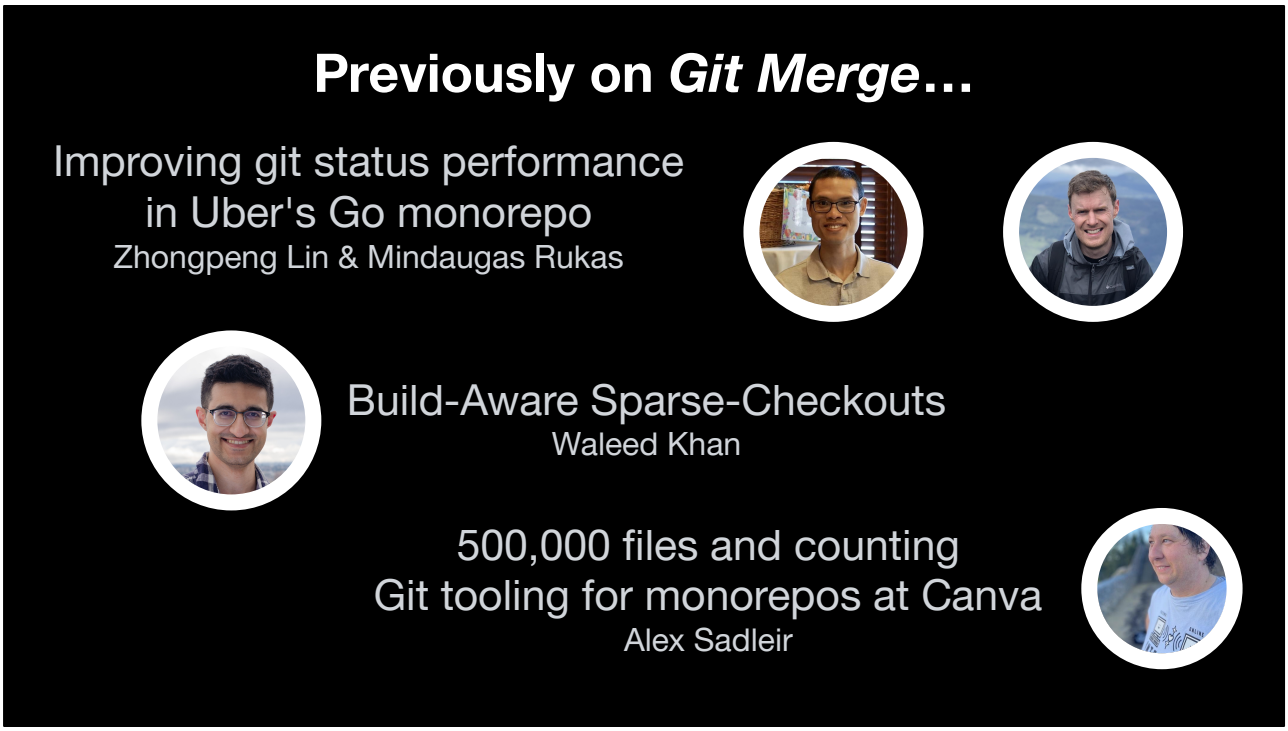


This leads to engineers learning a lot about how application databases work. There are thousands of books about how databases work and how you can write applications that take advantage of their strengths. No one bats an eye at the work required to understand your database and to interact with it properly.

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.



We've already seen three talks today about companies investing heavily in their Git infrastructure to ensure that it works exactly for their needs. These talks from Uber, Twitter, and Canva show what advancements can be made with that kind of investment.

(pause)

But we don't all have an engineering system so large that we can make that kind of investment. Many organizations need to rely on the Git client out-of-the-box as well as whatever Git hosting service they choose.

As a Git contributor, my main mission is to help every Git user achieve their highest collaboration potential. The Git community is constantly improving the Git client to meet these needs. Some times, these changes are immediate upon upgrading, but others require the user to opt-in to a new feature.

That's where this aspect of learning about Git like a database comes in: the more you know about your tools, the better you can use them.

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.

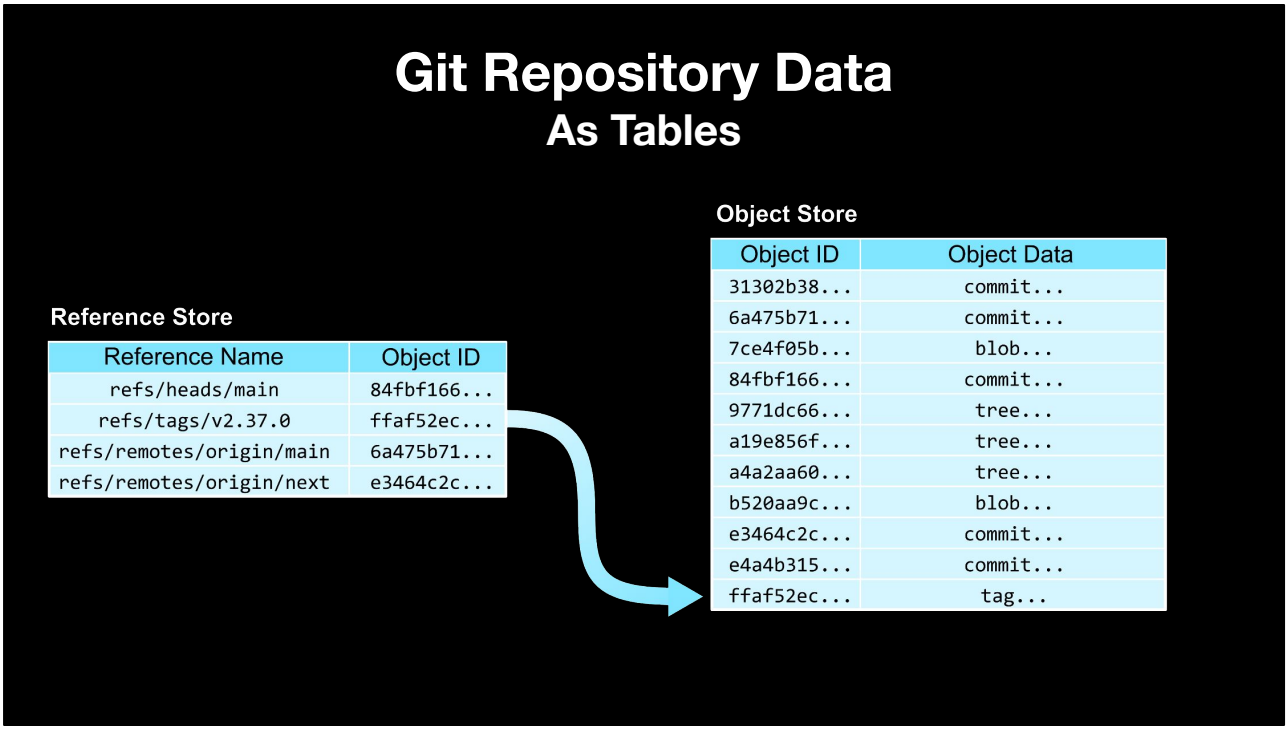
(click) Finally, when databases need to scale beyond the limits of a single node, application developers can use one of many sharding strategies. Git has similar strategies using multi-repo organization.

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.



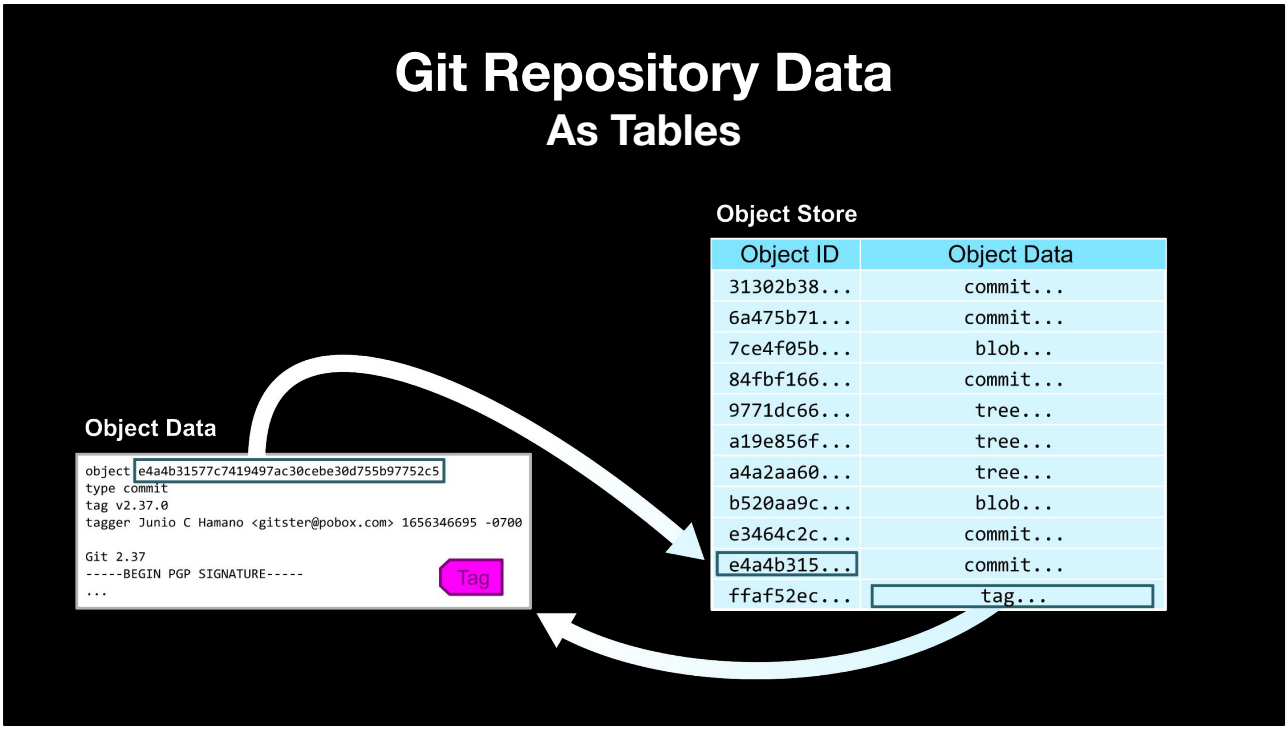
The main storage of Git repositories can be considered as two database tables:

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.

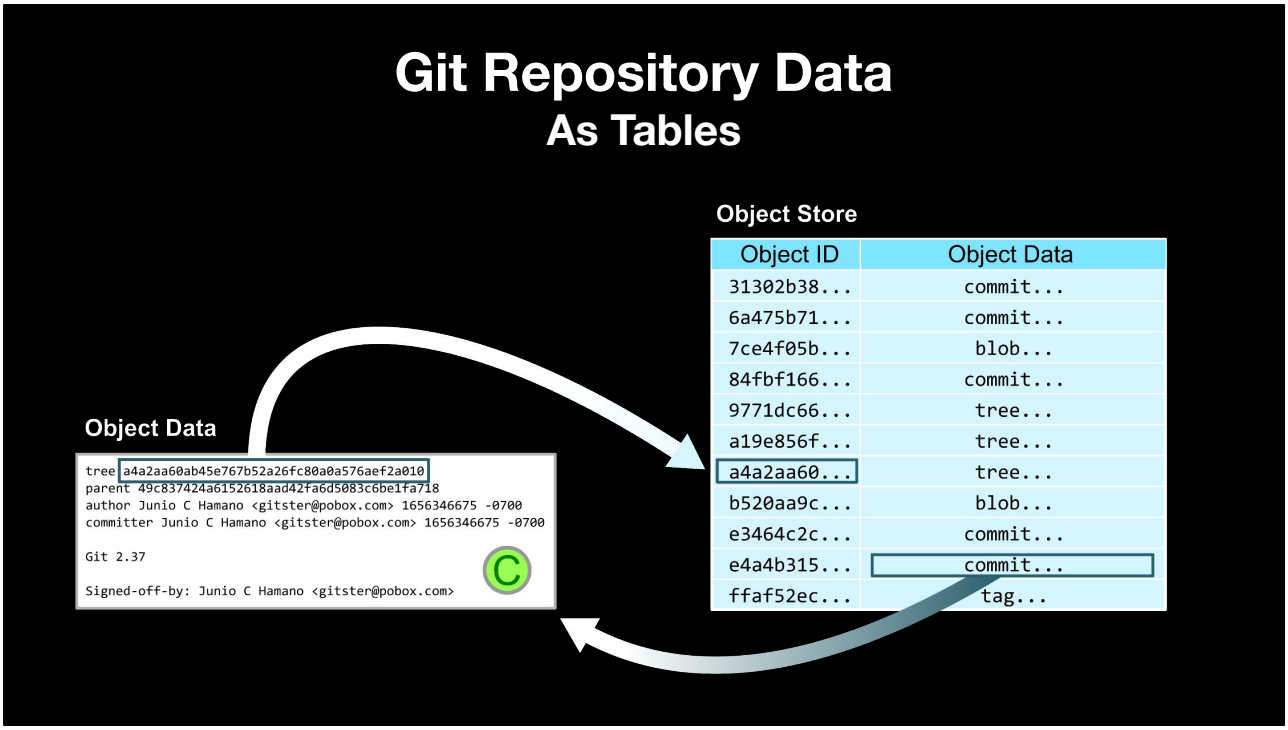
The reference store table has two columns: a reference name and an object ID. The reference name is the primary key. These human-chosen names are pointers into the opaque object store, allowing us to gain a foothold into the contents.

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



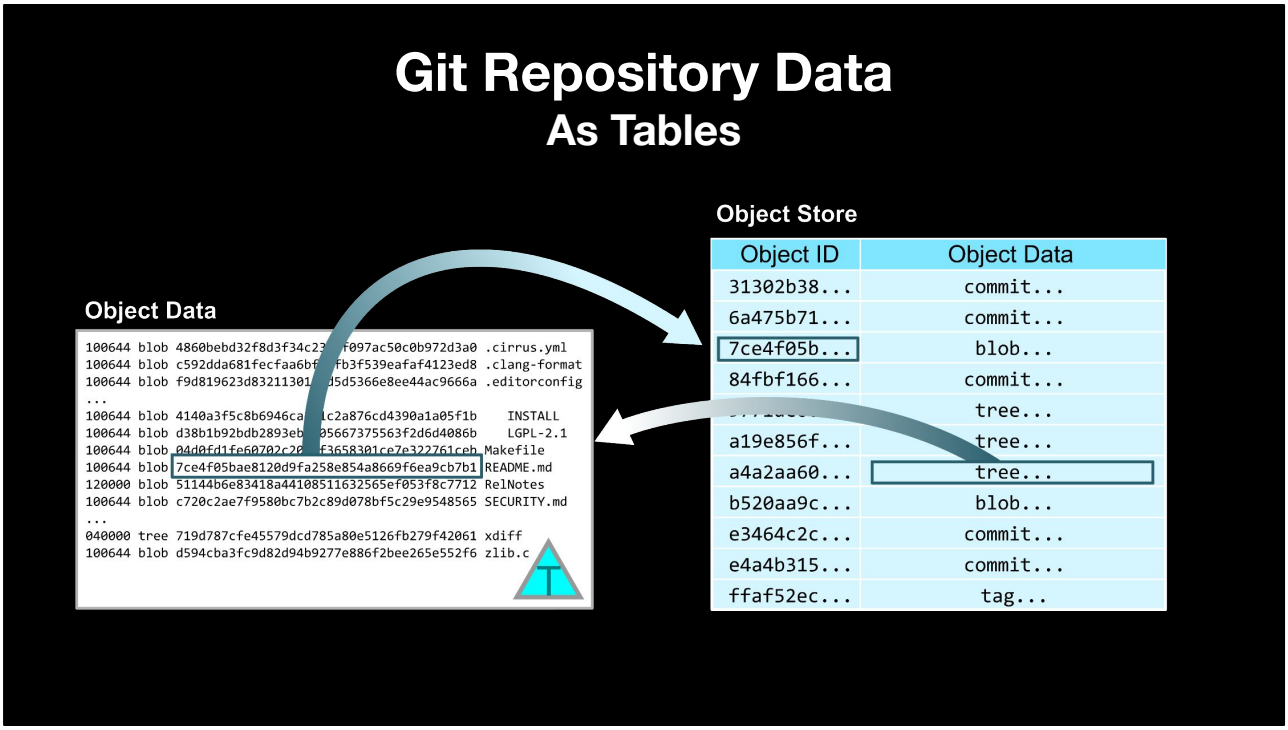
By pulling out the contents for that object, we see an annotated tag, which points to another Git object.

If we pull out that object and do a lookup in the object store, we see a commit object.



The contents of that commit object also includes a commit message, but it also contains a “tree” pointer.

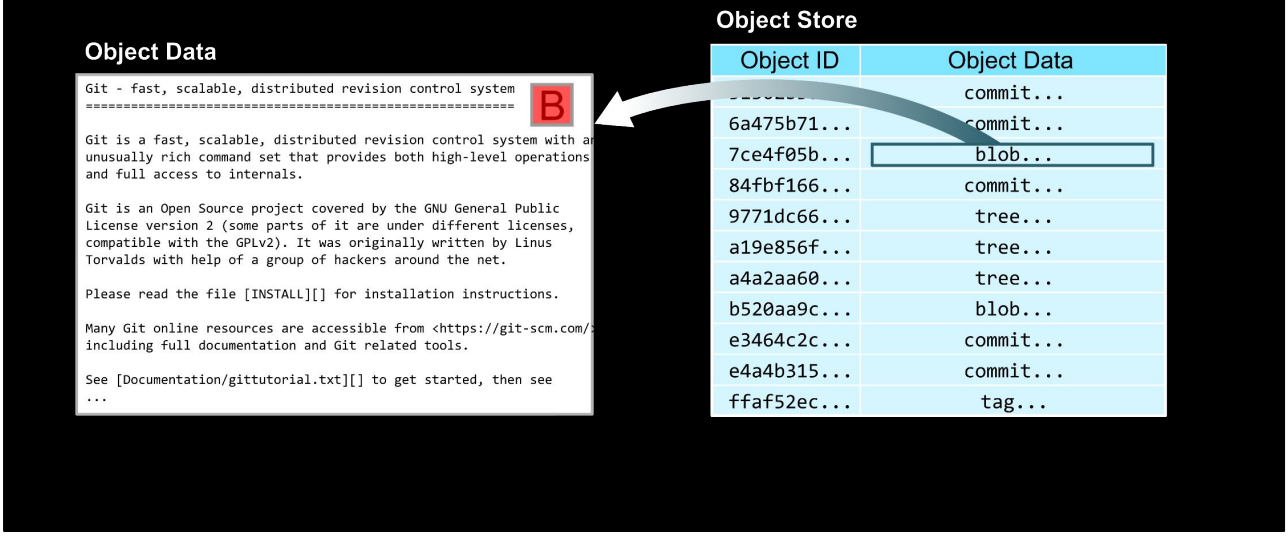
That tree object ID again refers to an object in the object store.



We pull that content out of the object store and see a tree object, containing several tree entries describing the root directory of a worktree.

If we focus on the tree entry for README.md, we find another object ID.

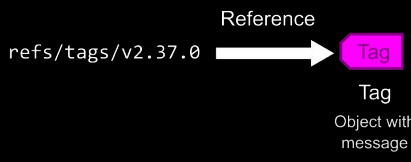
Git Repository Data
As Tables



And finally, that object ID is associated with a blob object storing the contents of the README.md.

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

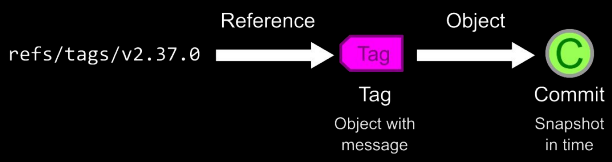
Git Repository Data
As Graph



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

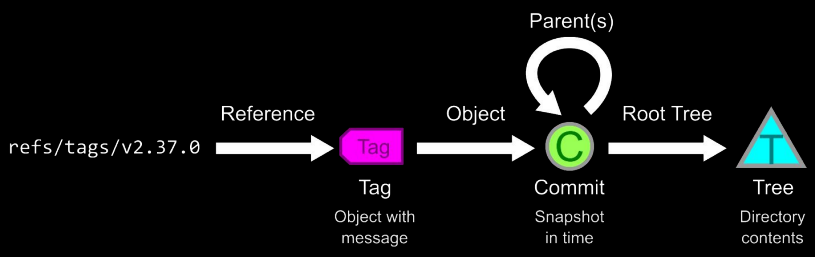
First, the reference points to a tag object.

Git Repository Data
As Graph



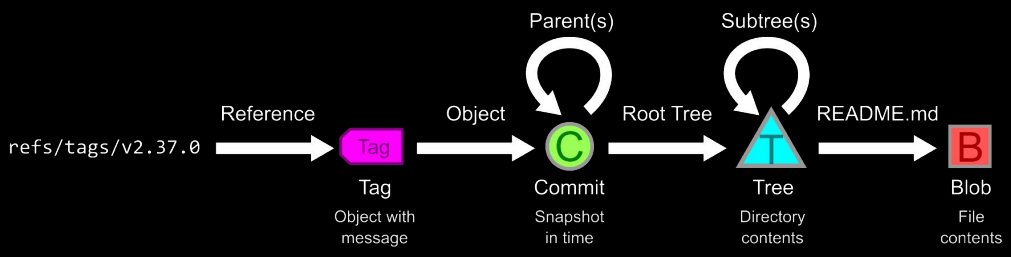
The tag object has an object pointer, pointing to a commit in this case.

Git Repository Data
As Graph



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

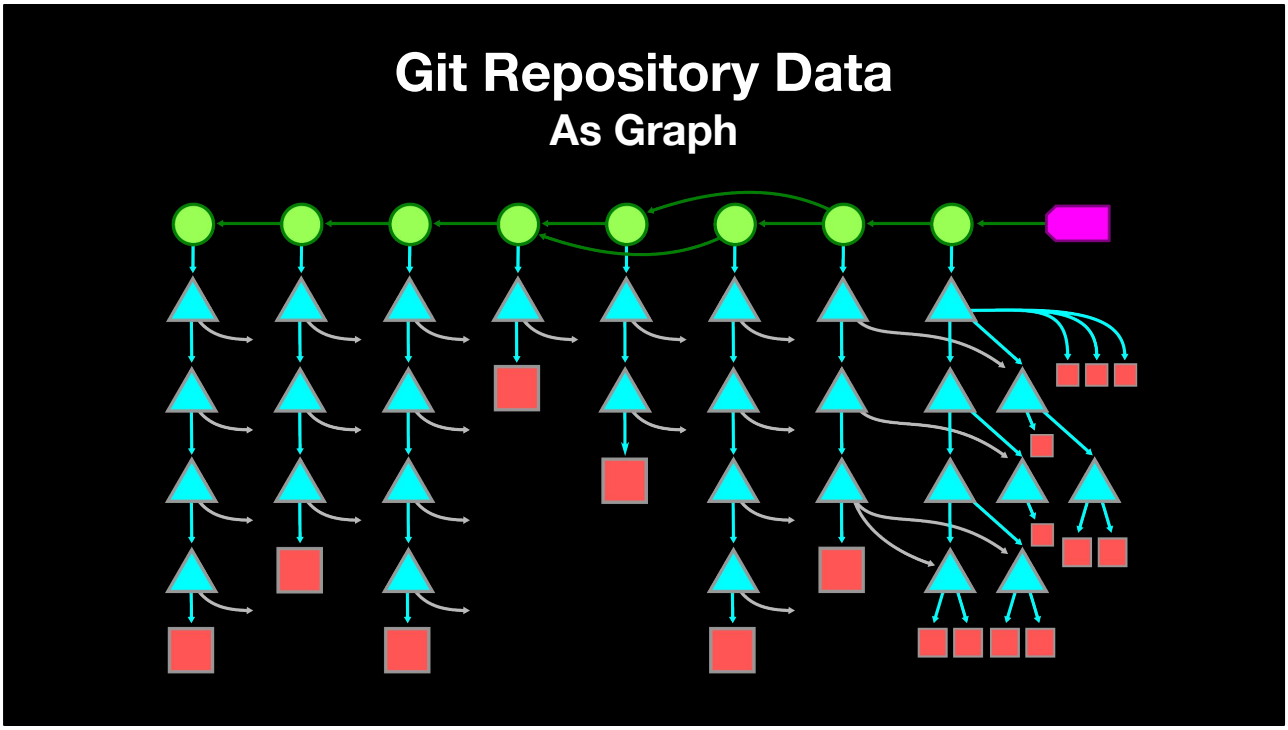
Git Repository Data
As Graph



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.

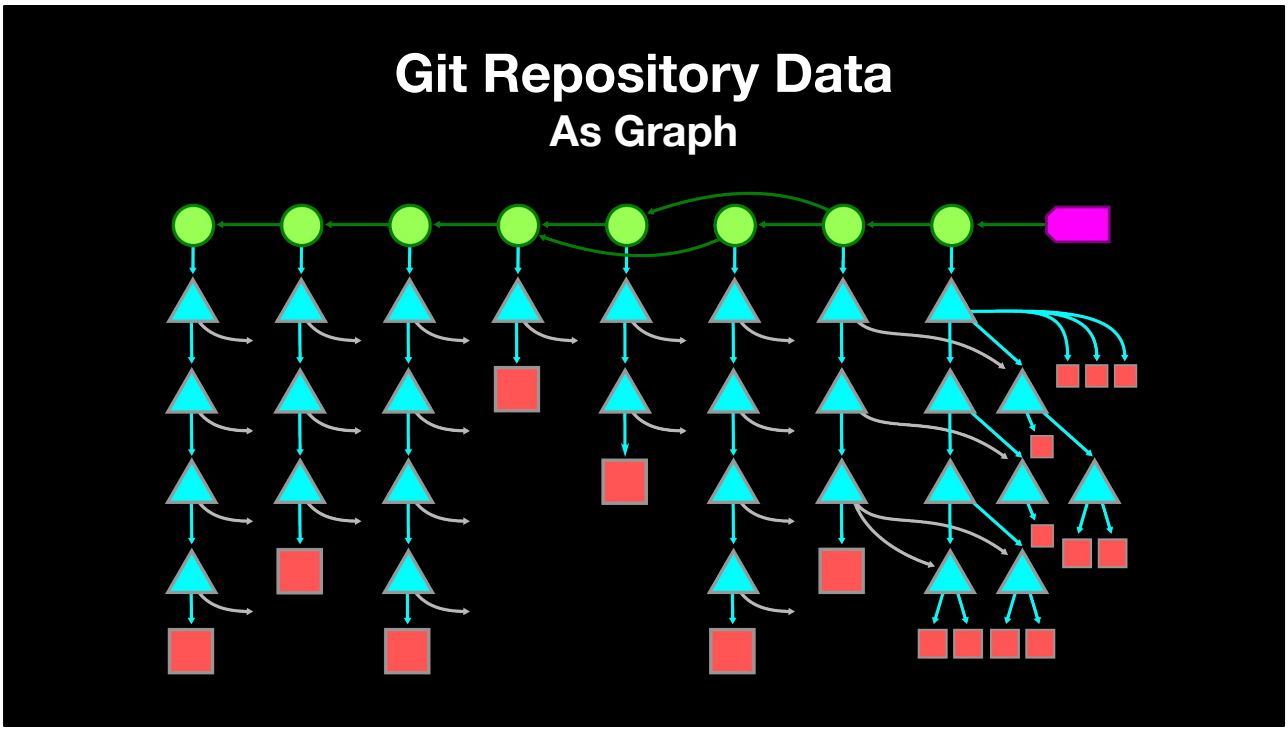
Previously on *Git Merge*…

Learn How Git's Code Works

Jacob Stopak

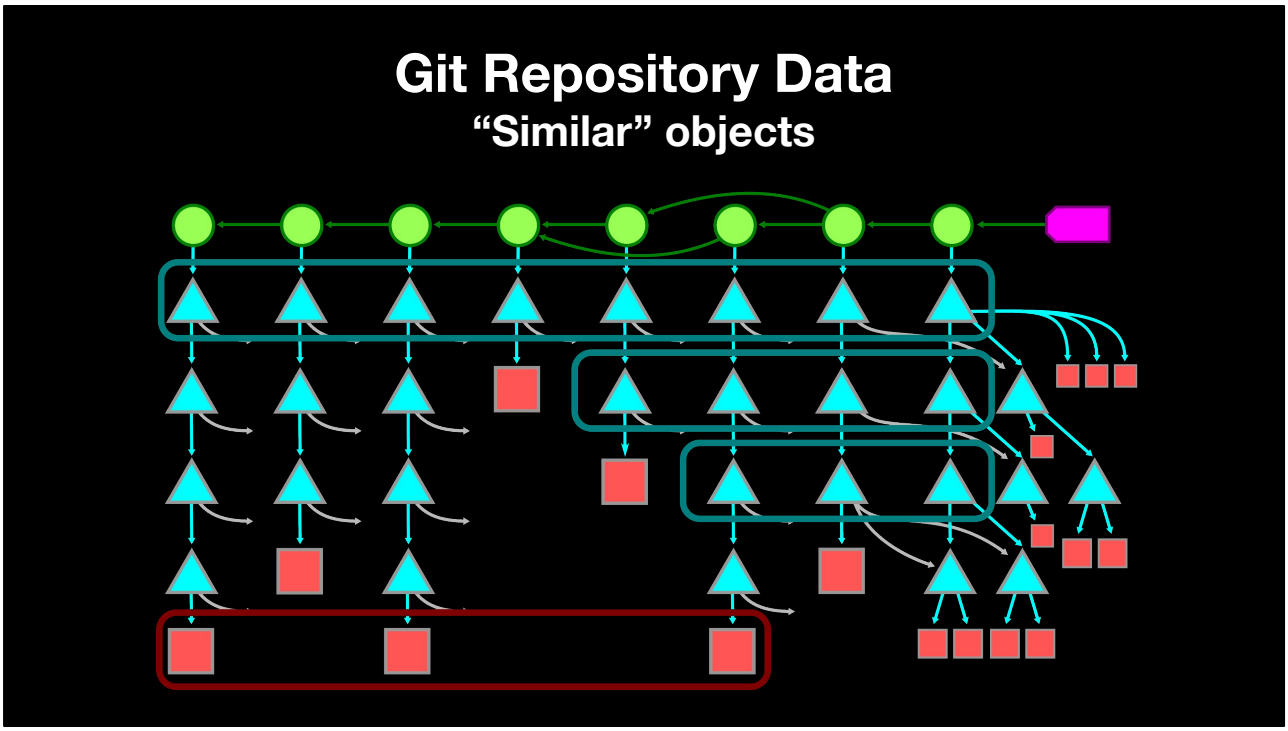
We heard a bit about Git's object store earlier today in Jacob's talk about how Git's code works, especially when it first started.

Jacob talked about how Git stores objects in their loose format where each object is written to a file whose name is based on the object ID.



When we look at the object graph and imagine it at full scale, we can imagine that loose objects will fill the filesystem too quickly.

One reason is the number of files themselves, but also loose objects cannot take advantage of a critical data shape in git repositories.



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

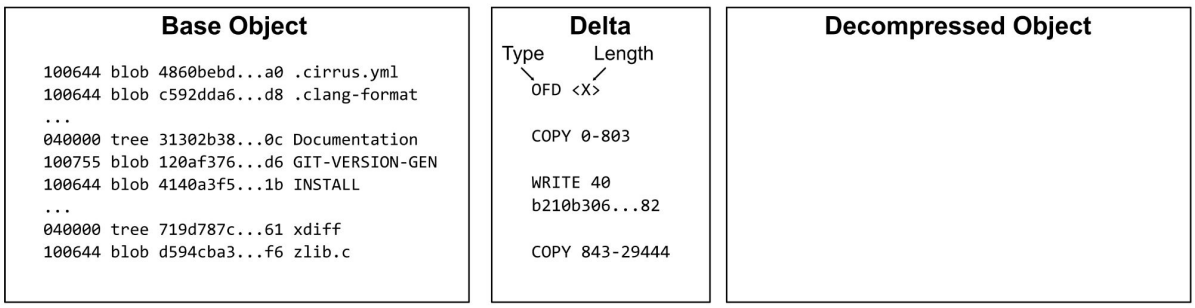
The top row of root trees all represent the base directory of the worktree.

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.

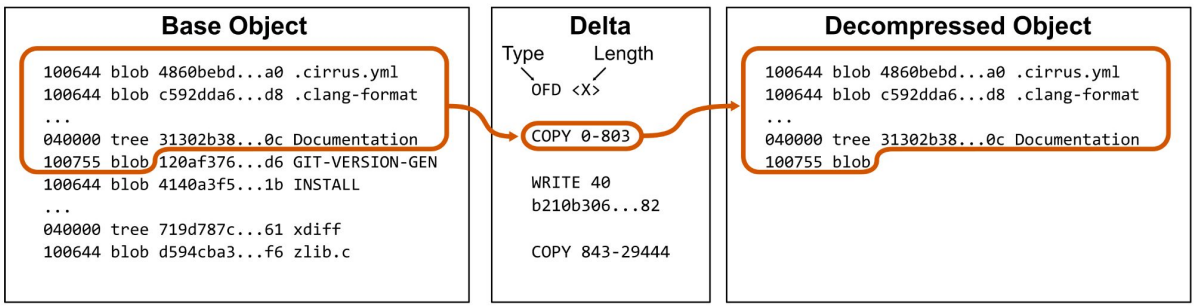
Object Storage
Delta Compression



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

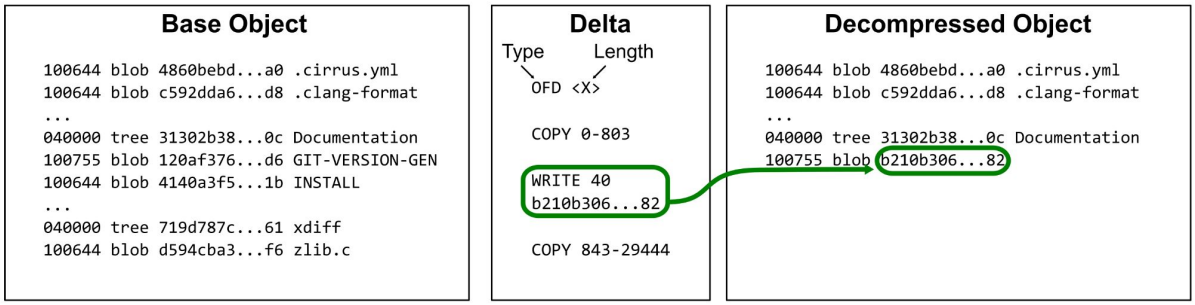
Deltas are instructions to help construct an object based on copying regions from a base object and writing new regions at certain points.

Object Storage
Delta Compression



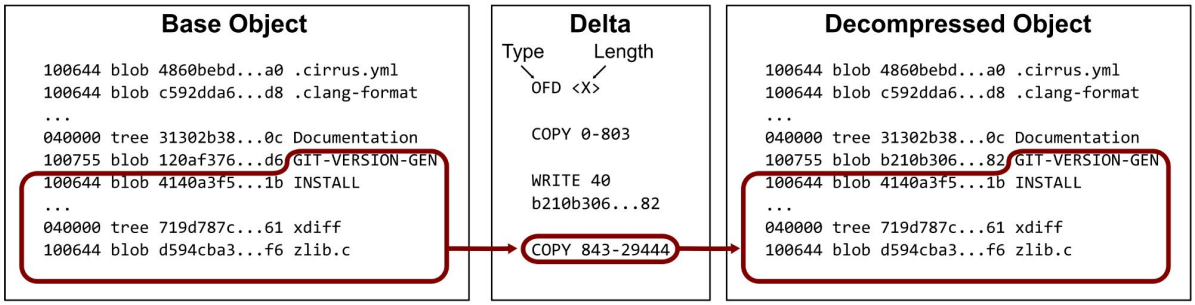
We start by copying the initial segment of the tree up until the object ID of the GIT-VERSION-GEN entry.

Object Storage
Delta Compression



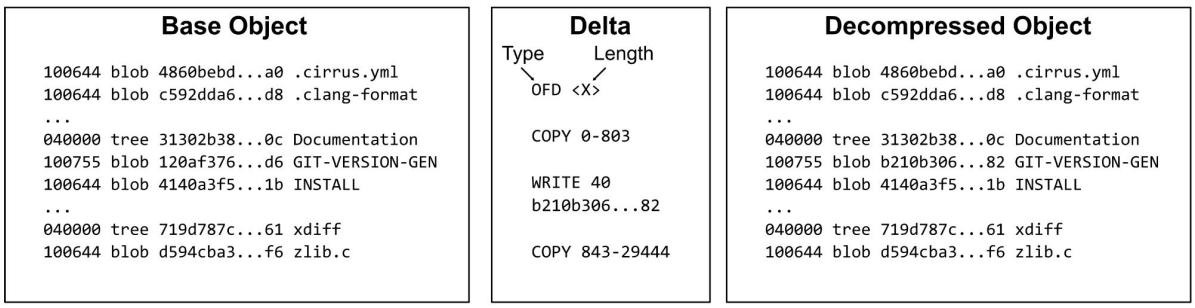
Then, we write a new section of data corresponding to a different object ID.

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.

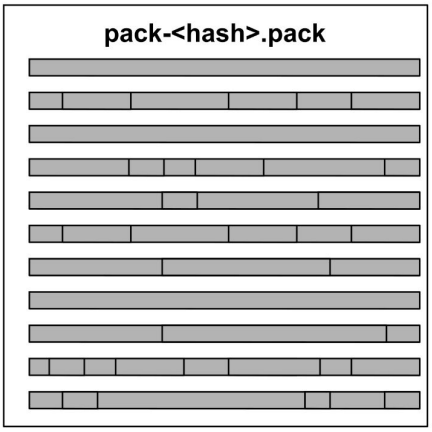
Object Storage
Pack Files

As I mentioned, loose object files cannot take advantage of delta compression. We need to use a different file format.

Object Storage
Pack-Indexes

Input
Object ID

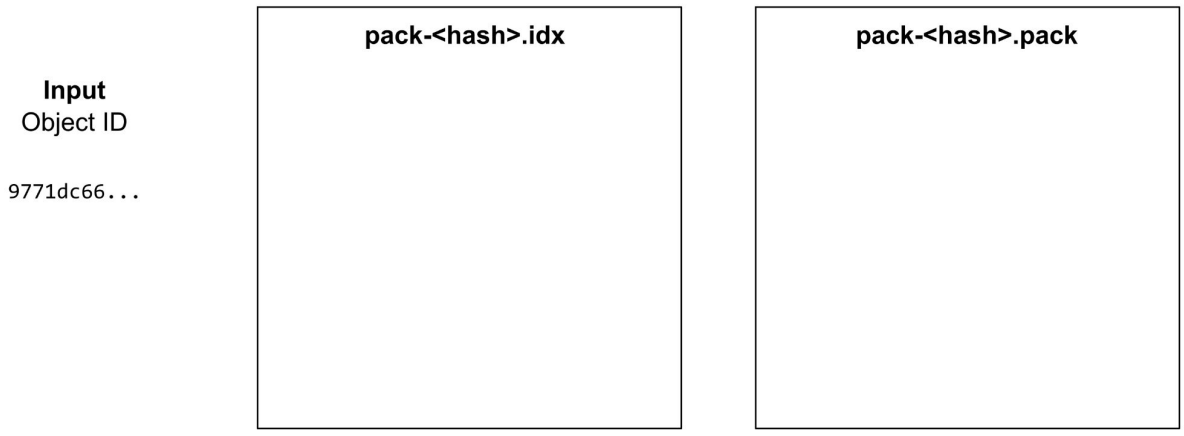
```
9771dc66...
```



The pack-file file format basically takes a list of objects and concatenates their content (They are “packed” together). In this format, objects can be represented as deltas of previous objects.

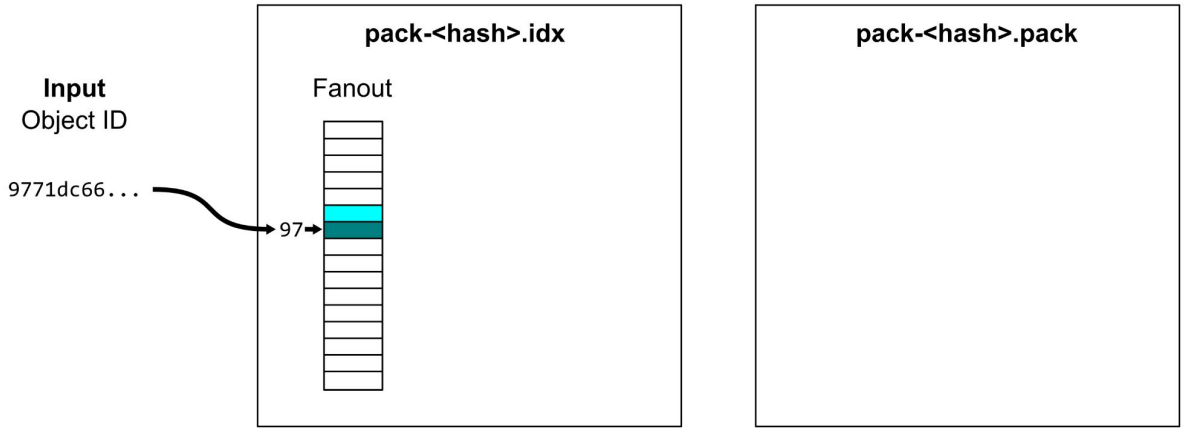
If we want to do an object lookup to find the contents for an object ID, the pack-file itself is difficult to parse. It would be too slow to scan the pack until we find a matching object.

Object Storage
Pack-Indexes



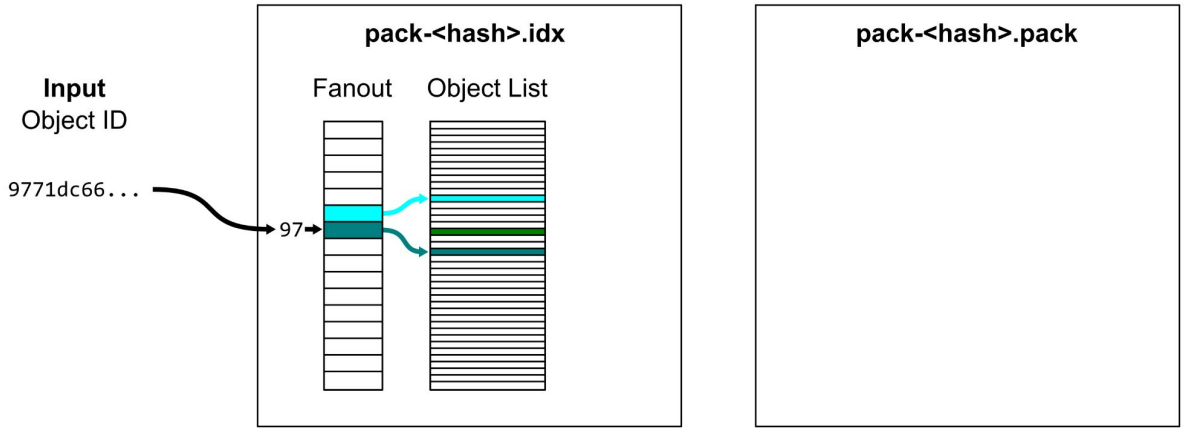
Instead, Git creates a query index called a “pack-index”. This is a .idx file to pair with the .pack file.

Object Storage
Pack-Indexes



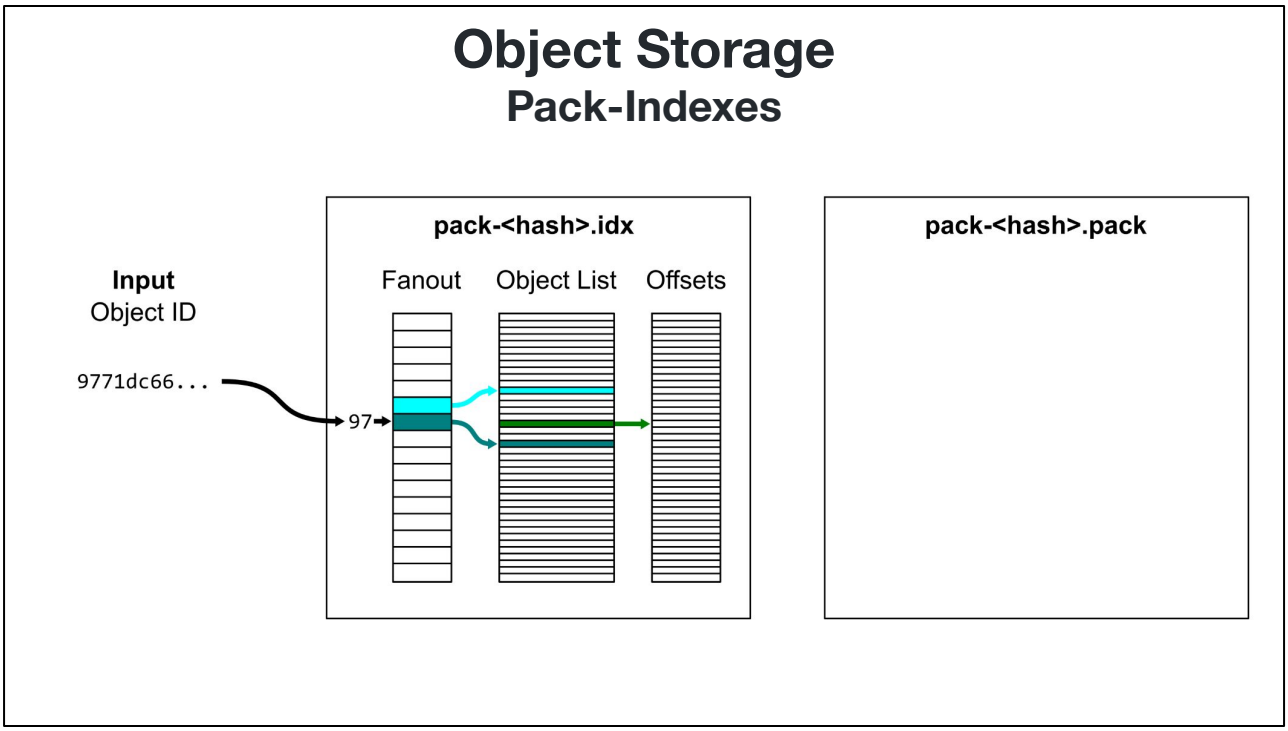
When doing a lookup, the first thing is to take the first byte of the object ID and look at that position of a 256-entry fanout table.

Object Storage
Pack-Indexes

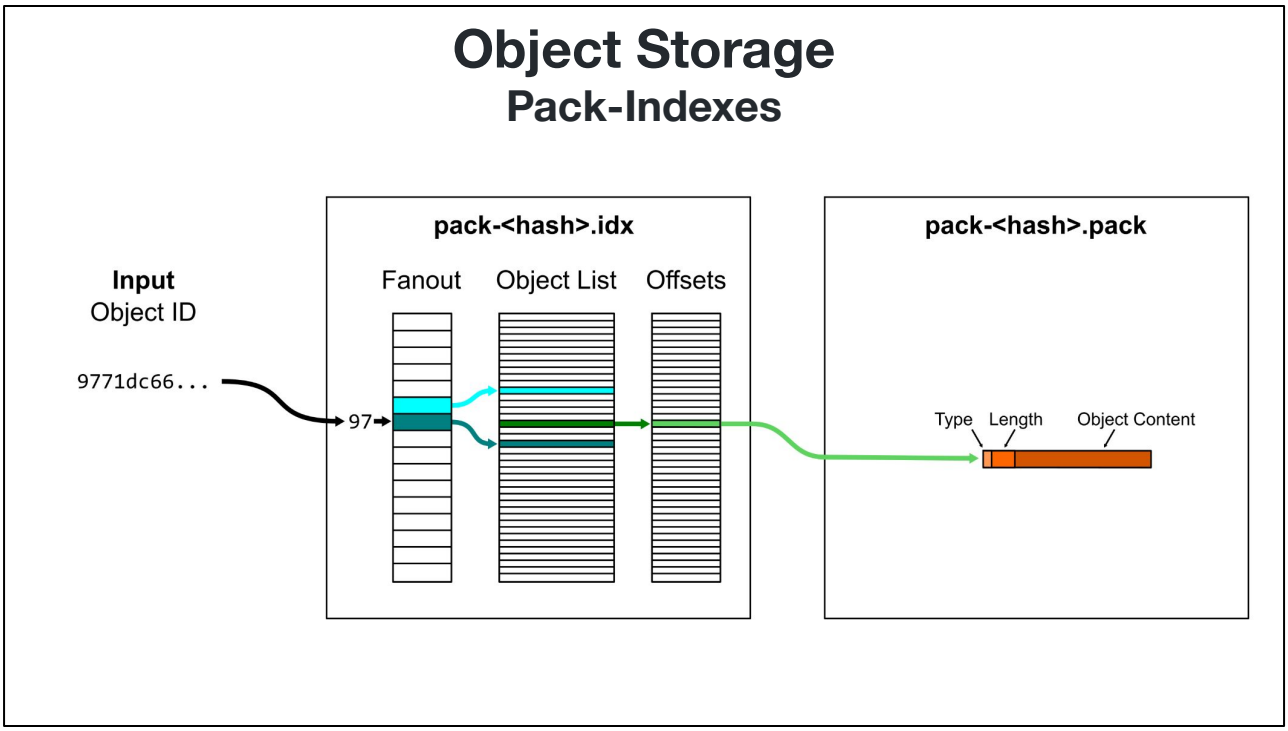


The fanout table provides a range of values in the object list, which is sorted lexicographically.

A binary search within that range can find the object ID.



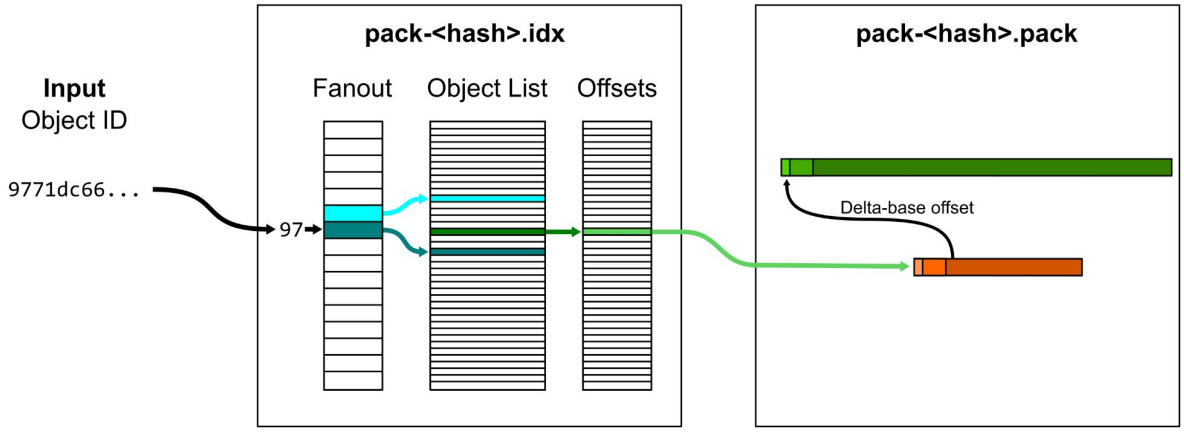
That position in the object list is paired with a row in the offset table.



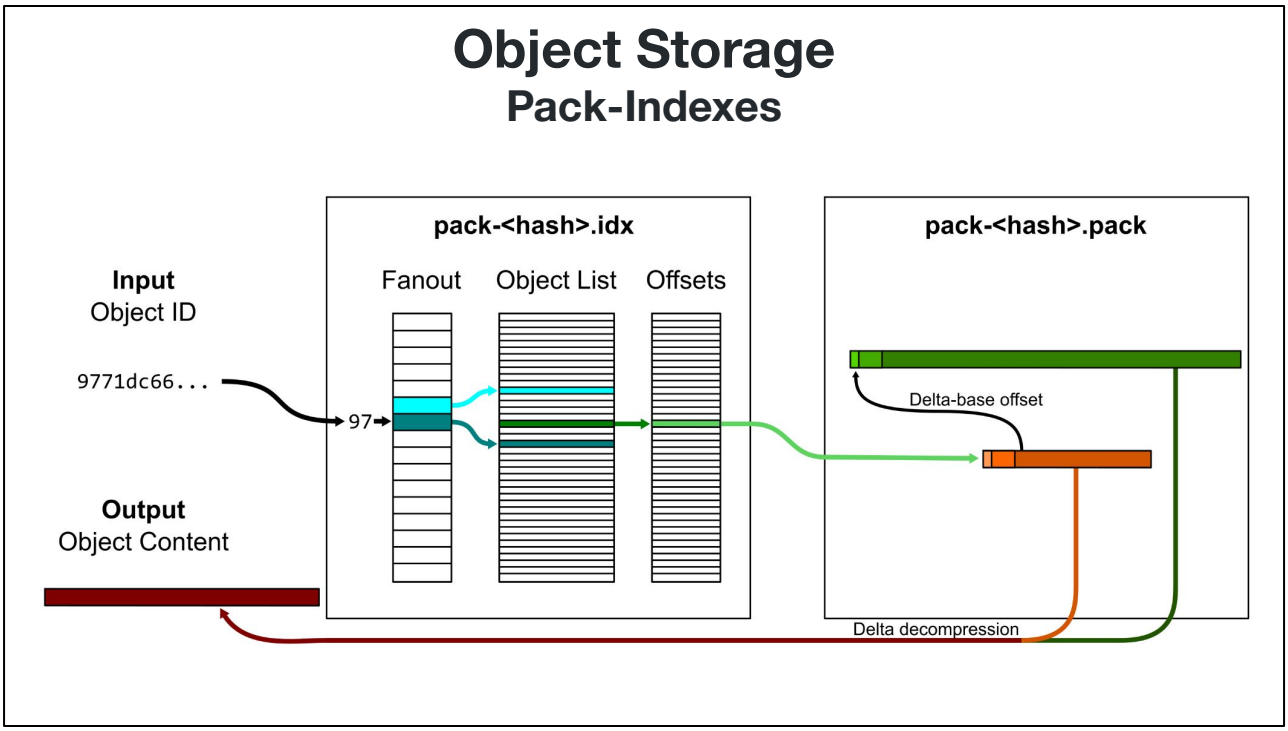
Finally, the offsets point to a position within the packfile where the object data starts.

The object data's initial segment includes a type and a length, allowing us to know how big the object is in the packfile.

Object Storage
Pack-Indexes



If the object is stored as a delta, then the content includes an offset to the base object that appeared earlier in the packfile.



Finally, Git performs delta decompression to construct the object content that matches the requested object ID, completing the query.

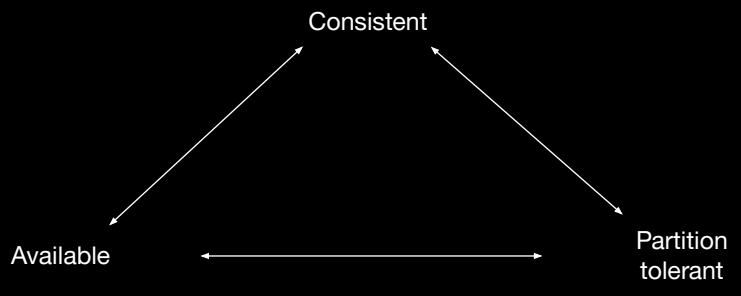
This object content lookup loop happens thousands of times as your Git commands process object data and follow links in the Git object graph.

Sending objects between repositories
Pack Files

But this is not the only use of the packfile format. 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.

Decentralized Git as Distributed Database



CAP Theorem

Remember that Git is a decentralized version control system, so every repository can act independently even as they coordinate together.

The closest parallels in application databases are distributed databases.

When thinking about distributed databases, the first thing we need to consider is the CAP theorem. A distributed system wants to be consistent, available, and partition tolerant. Unfortunately, these things are not possible simultaneously.

Distributed databases normally think about network partitions as an infrequent occurrence, allowing consistency and availability to be highly probable.

In Git, network partitions are the default state. Repositories communicate only when the user prompts for a fetch or push. There is no active synchronization when one repository changes.

This does mean that Git has chosen “available” and “partition tolerant” as its two modes with the CAP Theorem, which is a strength!

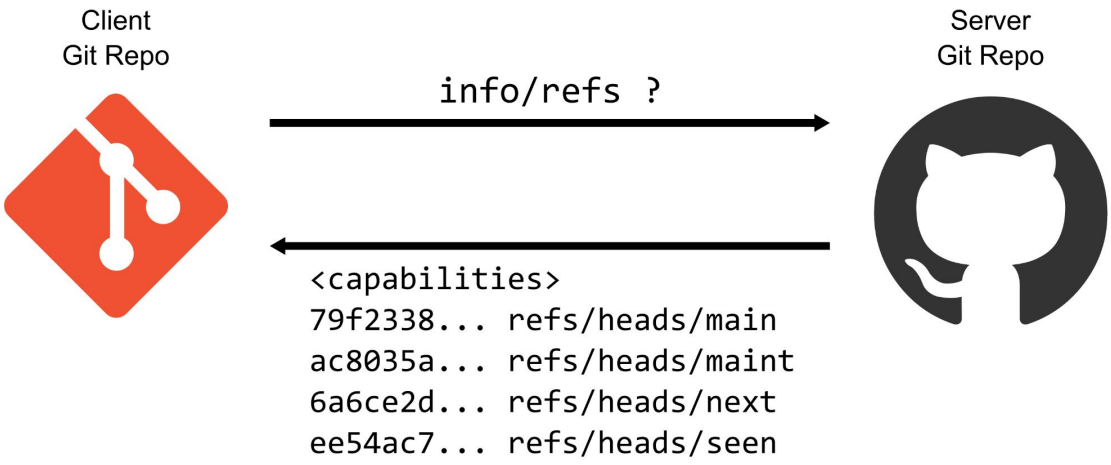
Repository Synchronization

Fetching new objects

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

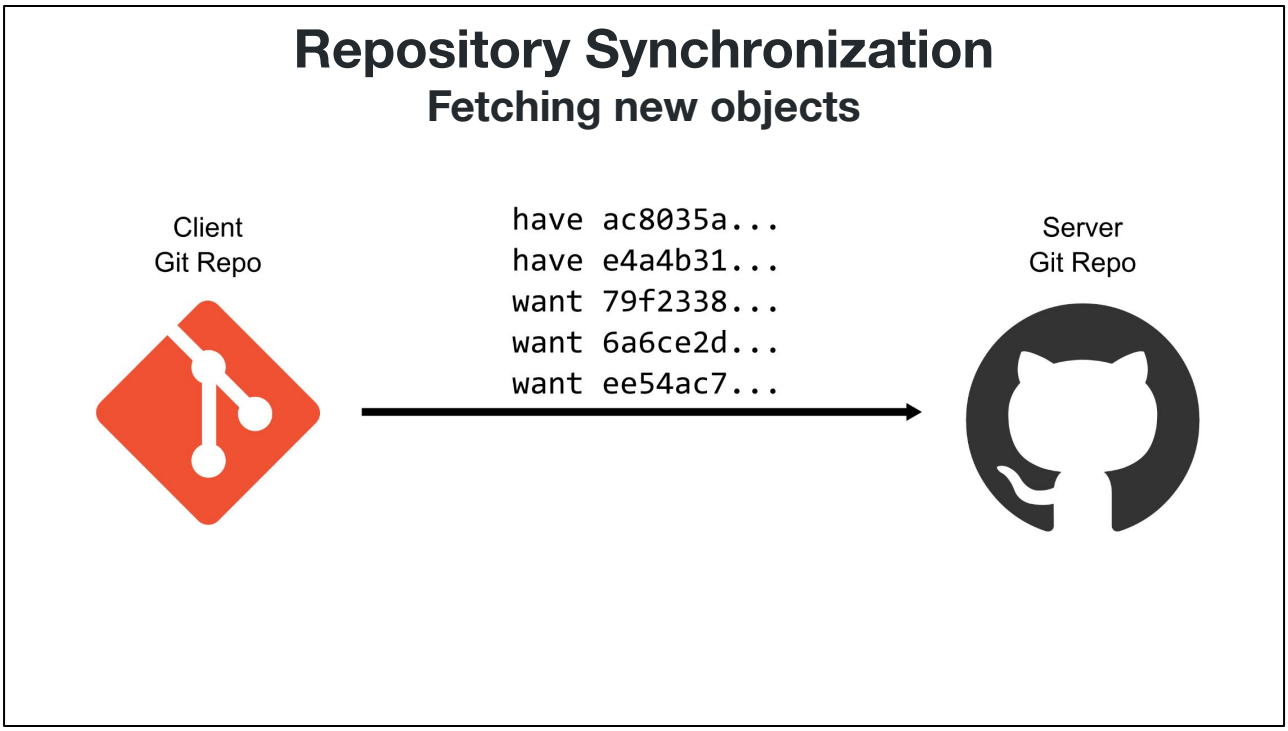
Repository Synchronization

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.



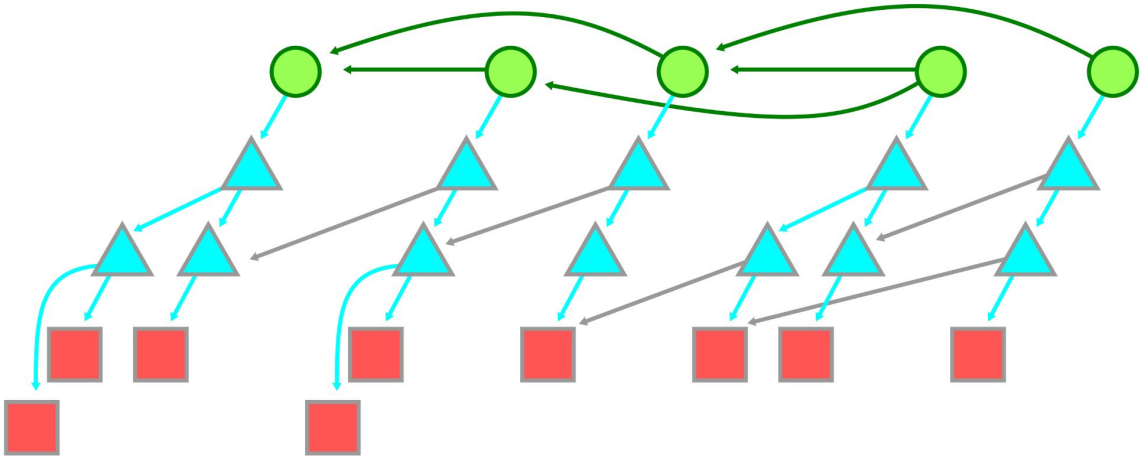
The client then sends a list of object IDs, each of which is marked as a “have” or a “want”.

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.

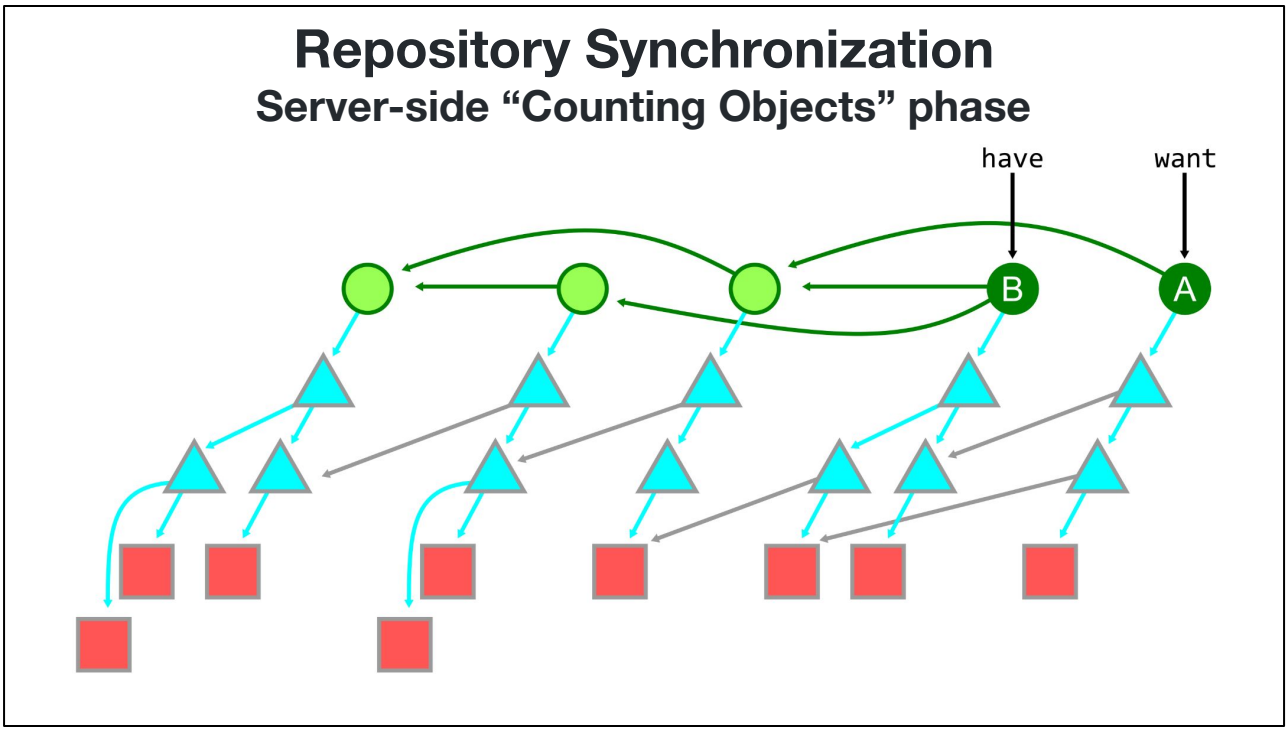
Repository Synchronization

Server-side “Counting Objects” phase

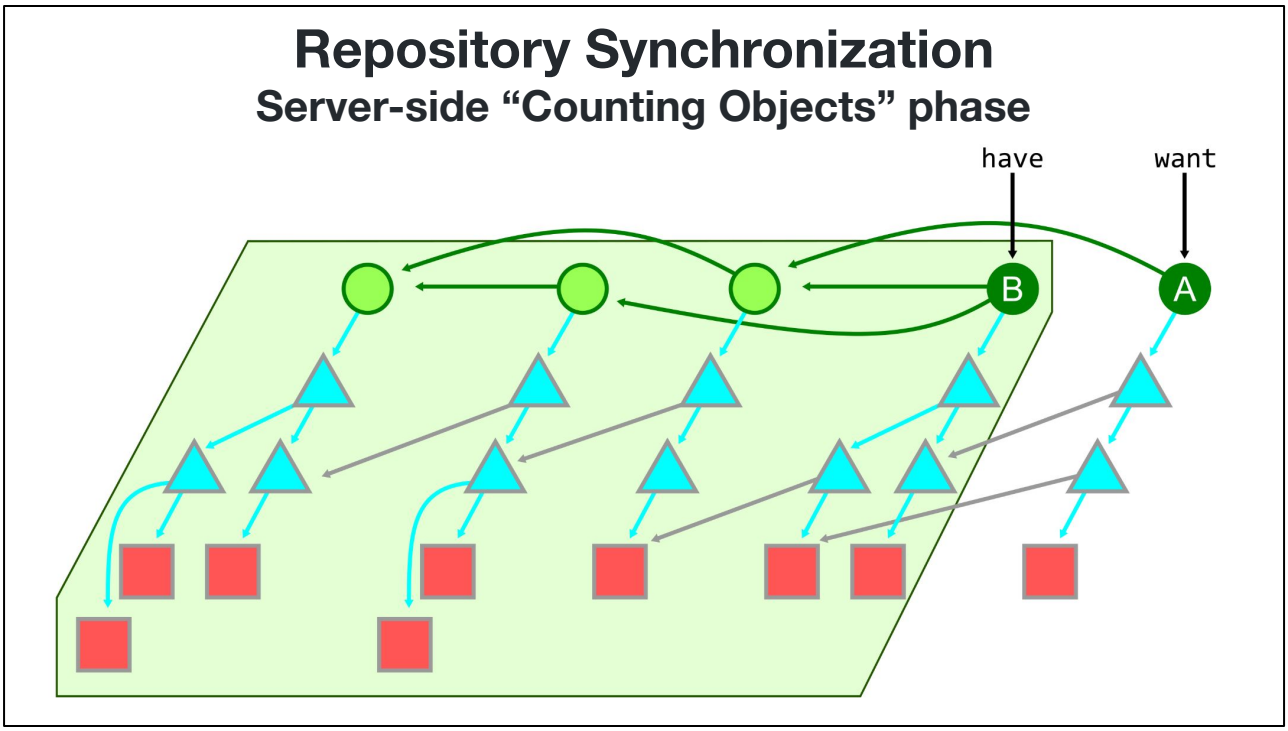


At this point, let’s take a look inside what the server is doing.

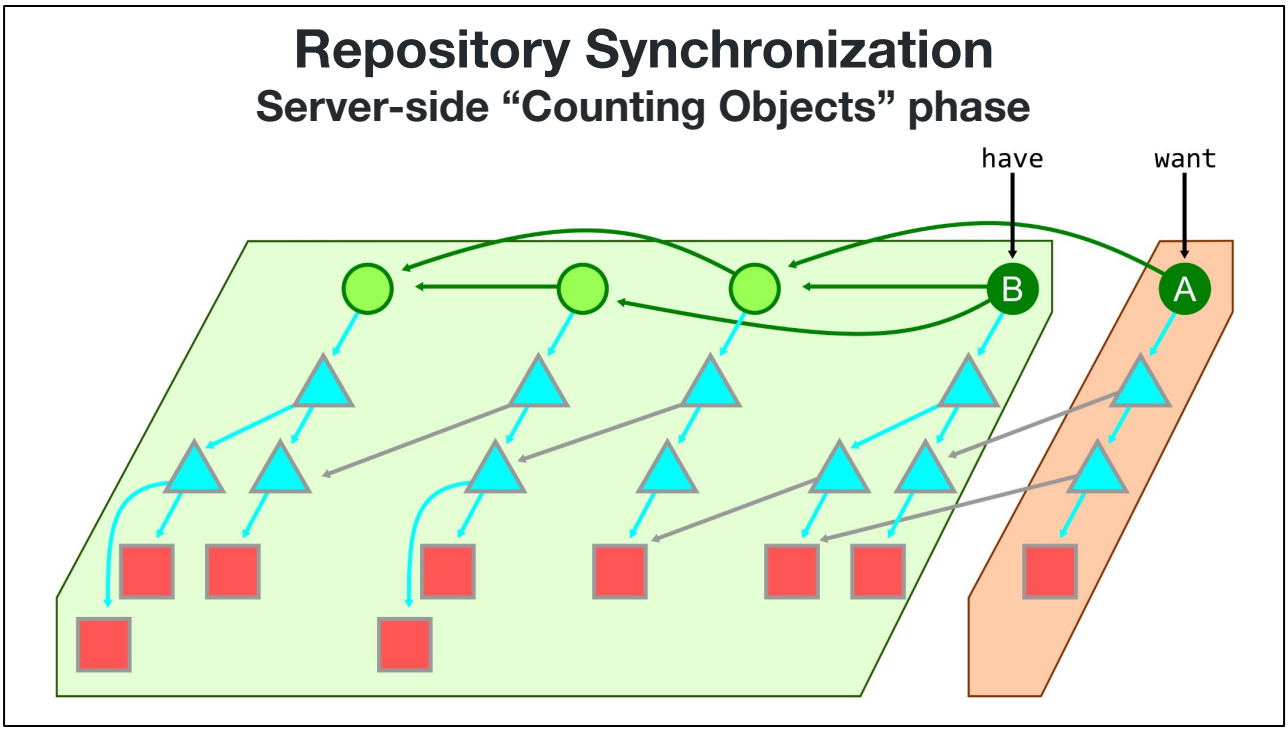
Here is an example object graph, including commits, trees, and blobs.



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



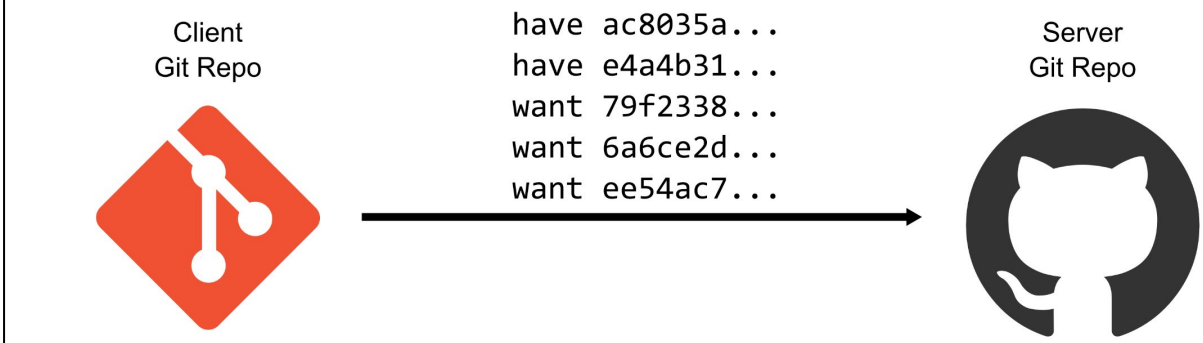
The server infers that the client has everything reachable from the haves, giving this region of objects.



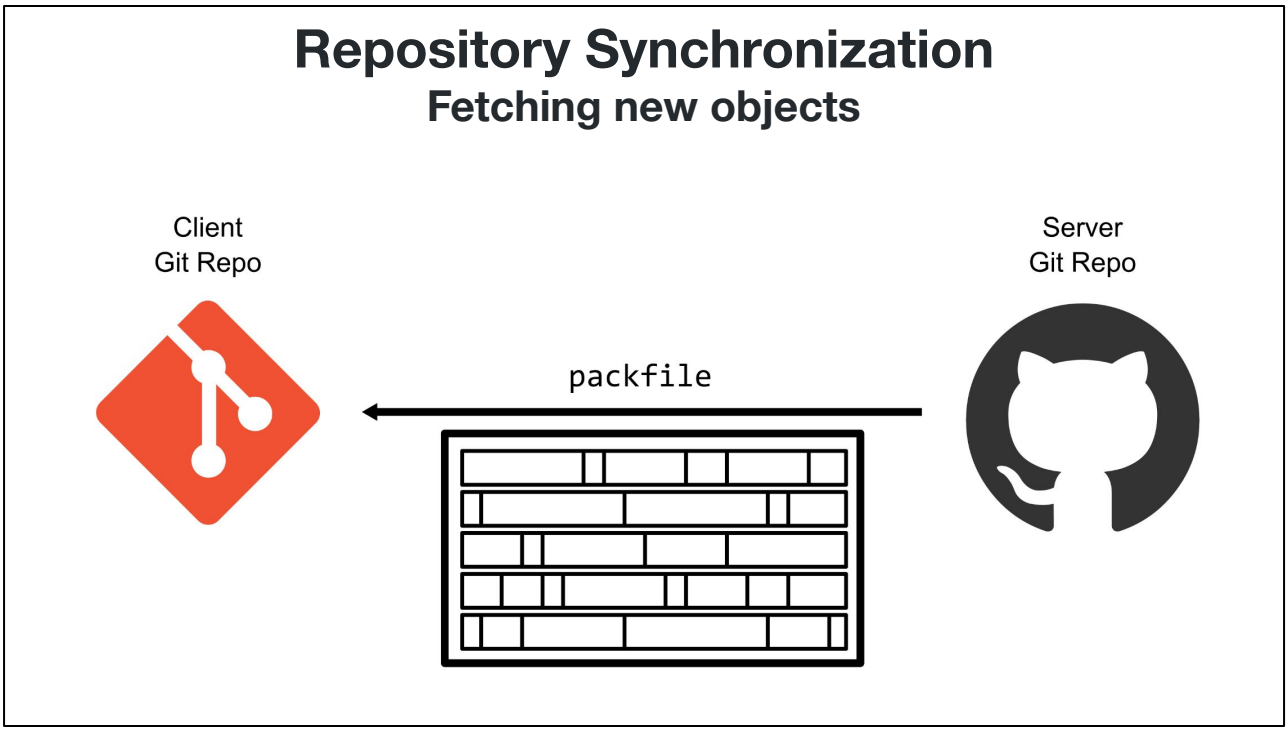
The server then determines which objects are reachable from the wants but _not_ reachable from the haves. These are the objects that the client needs.

Repository Synchronization

Fetching new objects



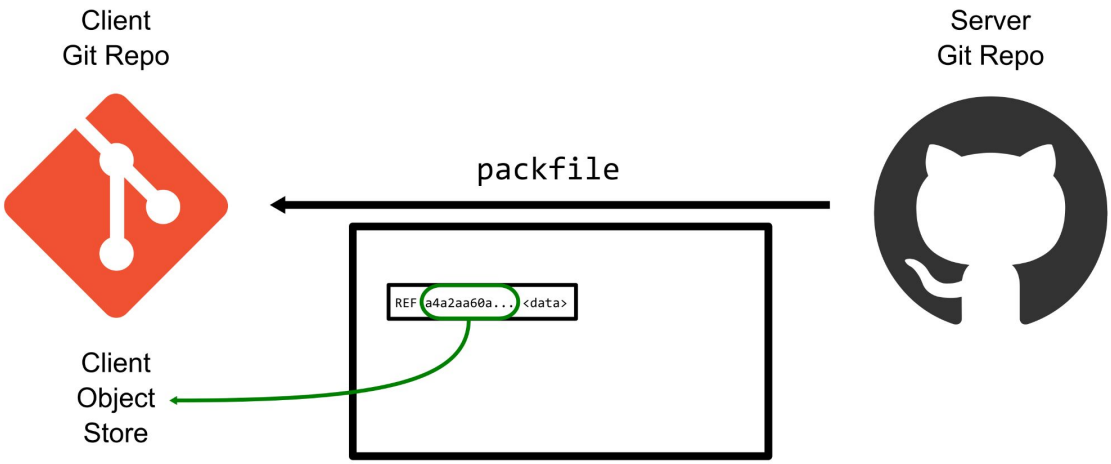
Back to the server/client interaction, the client may send a small number of haves and wants, but desires a possibly large set of objects.



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.

Previously on *Git Merge*…

Git at GitHub Scale

Taylor Blau

This operation of constructing a thin packfile based on a list of haves and wants, should be familiar from Taylor's presentation earlier this afternoon.

Taylor talked about how Git servers need to organize their packed object data as well as use custom query indexes like reachability bitmaps in order to quickly serve fetches and clones.

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.

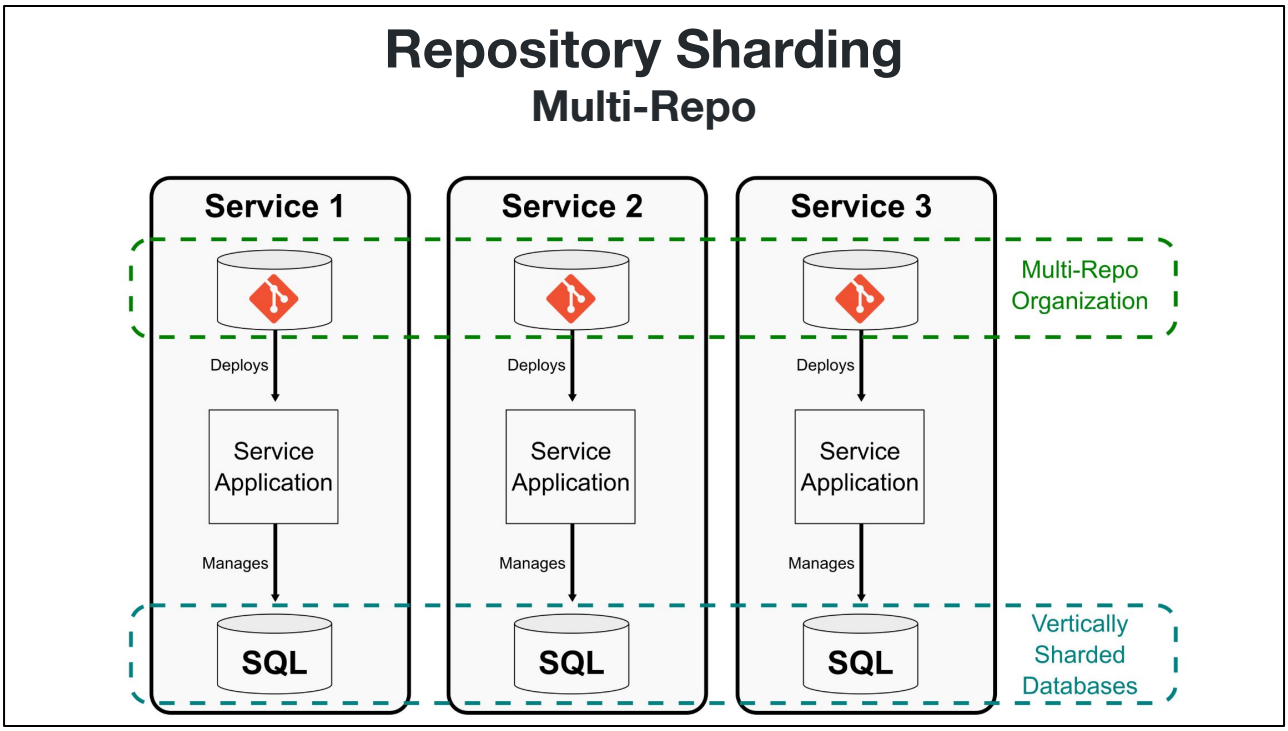
I've talked about monorepo scale a lot in the past, and it's been covered quite a bit today, so let's focus instead on these sharding strategies.

Vertically-Sharded Repository

Independent multi-repo

The first idea is vertical sharding, also known as functional sharding.

In Git, this corresponds to multiple independent repos.



This is typically done in a microservice architecture, where each service has complete control over its own application database layer.

In this world, the databases are sharded vertically by functionality, and the services are built, tested, and deployed from independent repositories.

This organization optimizes for small repositories and low bureaucratic overhead to rapidly deploy each service.

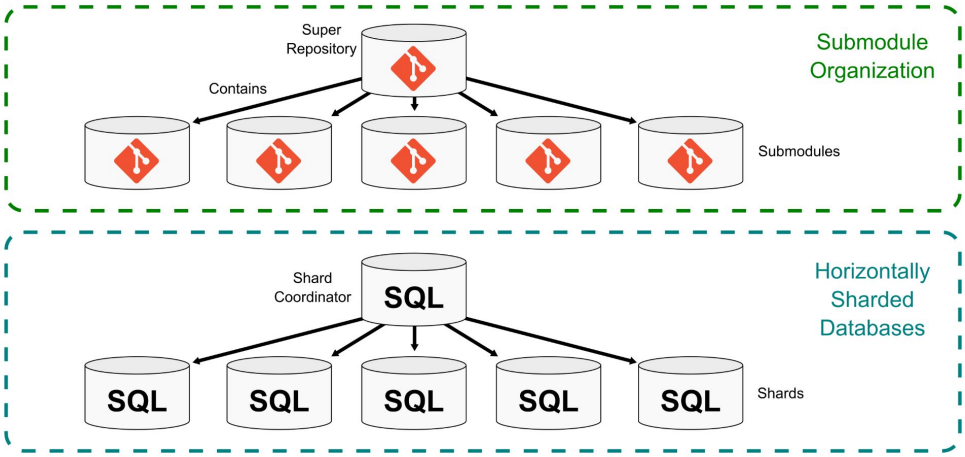
The problem comes in when there is no coordination between the repositories and services that group together to a larger whole. This can be particularly frustrating when trying to work across service boundaries because it is hard to find the service you need without some human overhead.

Horizontally-Sharded Repository

Super-repo split into submodules

This brings us to our next sharding strategy: horizontally-sharded repositories using submodules.

Repository Sharding
Submodules



A horizontally sharded database takes the same kind of data and spreads it across multiple database nodes, using a shard coordinator to help queries run on the proper shard nodes.

WIth submodules, a superproject repo has “git links” to the commit histories of multiple submodule repos. Those repos can move ahead independently, but eventually the superproject dictates which version each submodule is in when considered part of the whole. Each submodule is identified in the super project by a path prefix, which is the shard key.

This solves the coordination problem in that every submodule has a concrete link fron the super-project, so there is a clear way to find the necessary pieces of the larger whole.

Previously on *Git Merge*…

An Improved Workflow for Submodules

Emily Shaffer

But don’t take my word for it. Emily talked earlier about submodules, and if you have any questions, I will direct you to her as *the* expert in the space.

One thing I can say is that if you didn’t start with submodules, then it can be a lot of work to split sections of your repository into distinct pieces.

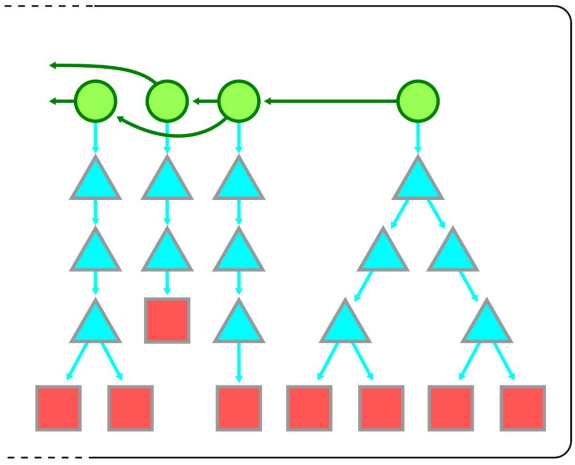
The next sharding strategy doesn’t require any changes to your worktree organization at the tip of your repository.

Repository Sharding
Time-based Shards

Time-based shards are inspired by time-series databases.

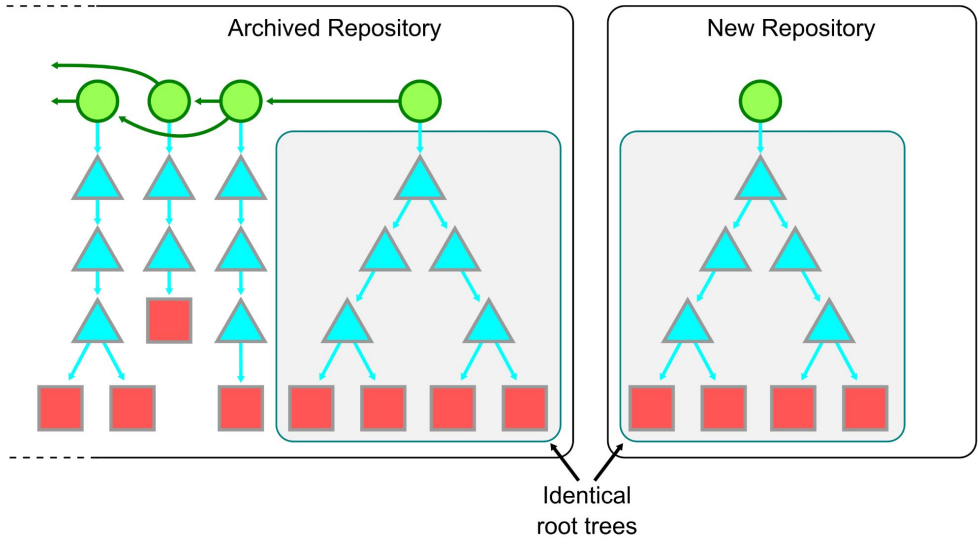
This presents a way to shard a monorepo without needing to restructure your working tree. It's particularly useful if you've gone through your monorepo and cleaned up the large binaries and removed the build output so you are practicing good hygiene, but want to rid yourself of the previous data shape in your repository.

Repository Sharding
Time-based Shards



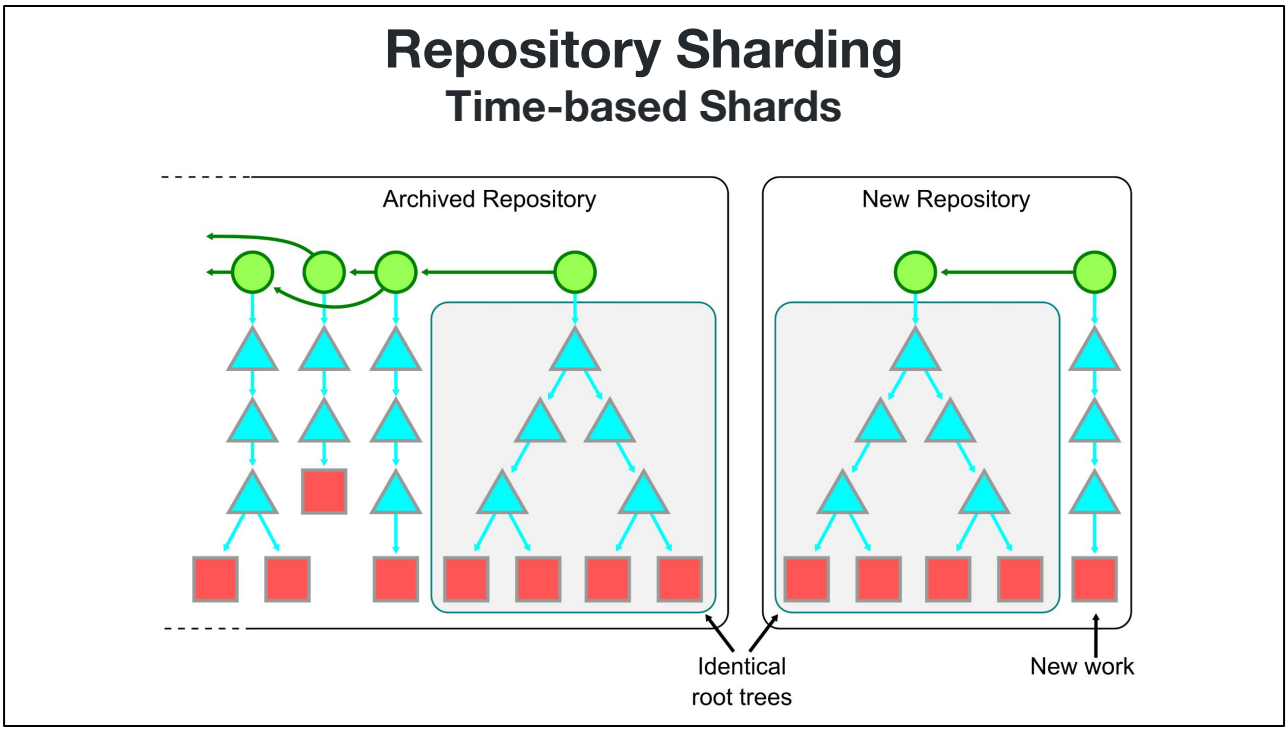
So suppose we have a large repository with a significant commit history, but let's focus on a single tip of the history.

Repository Sharding
Time-based Shards



We can archive the old repository and create a new one with a root commit whose root tree matches the root tree of the previous tip.

The objects in this commit may be large, but will be significantly smaller than the full object graph.

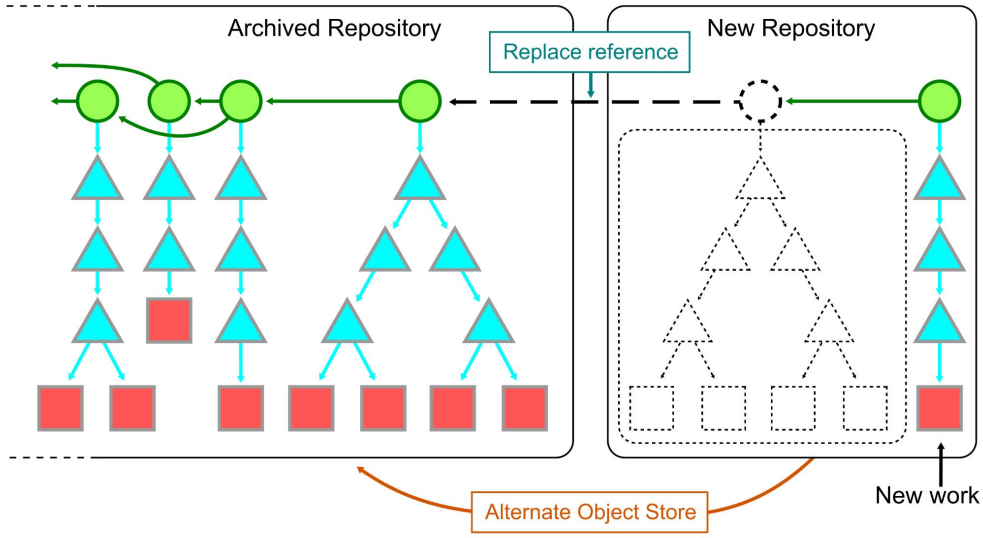


Now, new work can continue on that root commit, moving the new repository forward.

This process requires a lot of coordination to complete the migration. There is no “magic button” that does this for you (at least not right now).

The problem here is that the root commit looks like it added all of the objects in the repository all at once. The helpful commit messages from the previous history are no longer available to developers trying to solve issues in the new repository.

Repository Sharding
Time-based Shards



Git can help with this, by allowing a way to link the two repositories together using “replace objects”. By marking the root commit as replaced by the previous tip, Git commands can pass into the old history seamlessly.

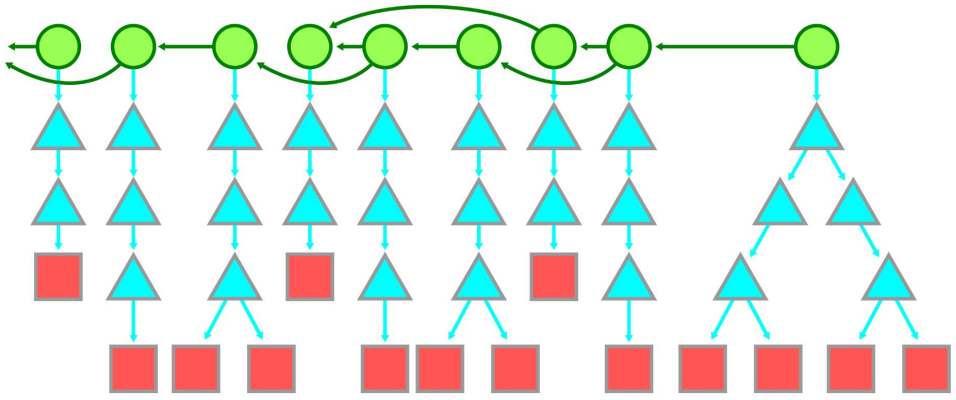
This is not a very efficient setup, so it is recommended that archived repository is available via a Git alternate and the replace objects are present only when needing to look into the old history. As work moves forward in the new repository, the need for such history queries slowly diminishes, but is always available if necessary.

Data Offloading
Prioritizing “important” objects

Finally, another organizational concept from application databases may be useful for Git, but is not exactly a sharding strategy: data offloading.

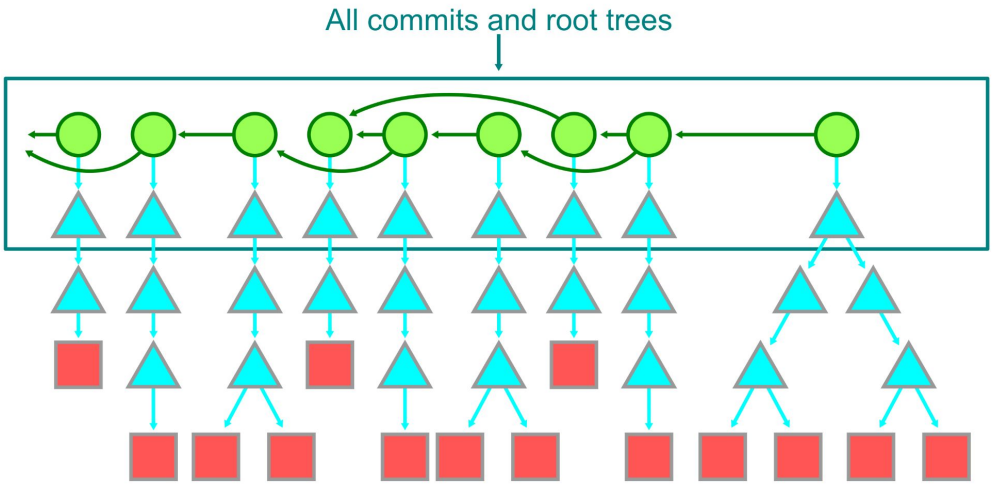
The basic concept is to take data that is accessed infrequently and offload it to cheaper storage.

Data Offloading



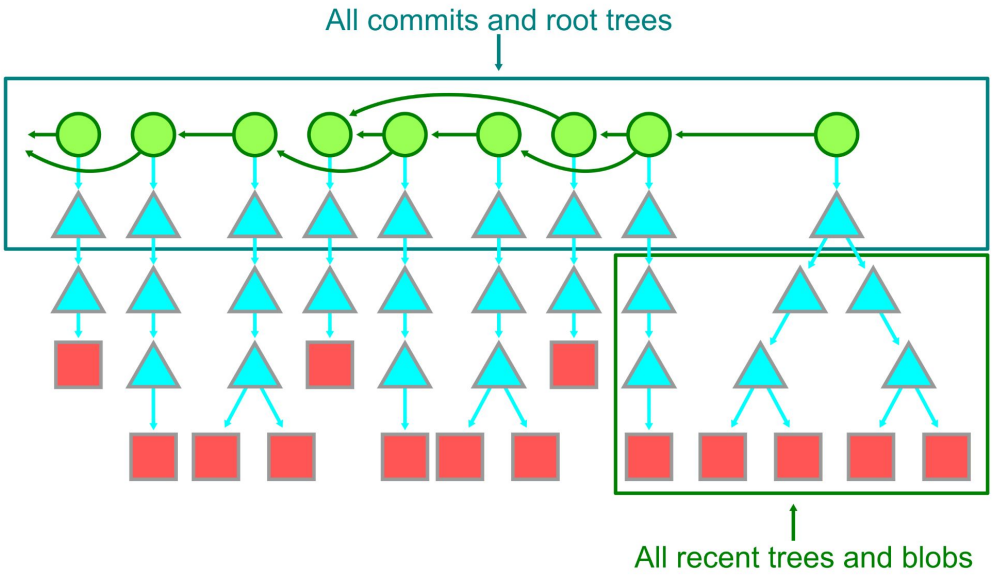
Let’s think about this object graph.

Data Offloading



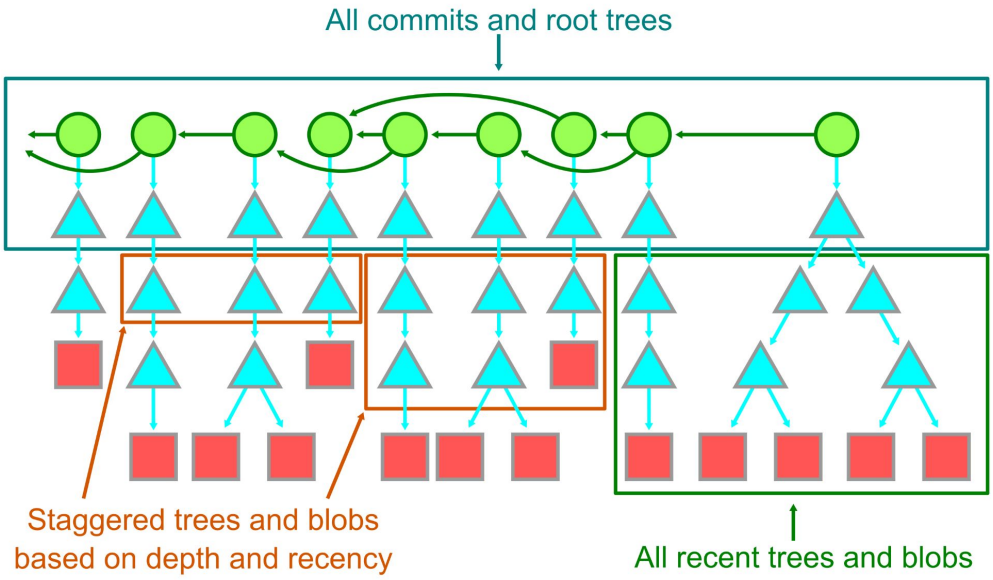
The commits are very small and easy to store, and they are really important to many Git commands.
Their root trees are typically also easy to store with delta compression, so all commits and root trees should be considered as critical data.

Data Offloading



The objects reachable from the root trees of recent commits should also be considered important.

Data Offloading



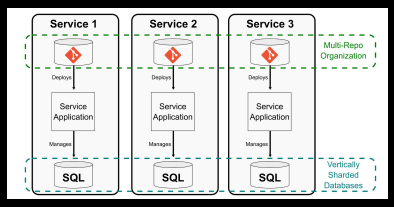
AS we go back in time, we can reduce the depth of objects that matter, since we are less likely to go deep into a given path the older the commit.

This creates a way to label objects as “new” or “old” such that we could keep only the new objects on local disk while having the rest of the objects available in an alternate over a read-only network share.

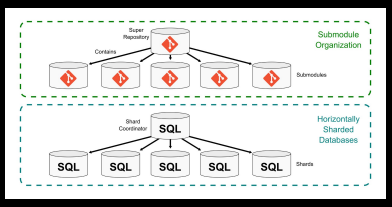
As far as I know, there isn’t currently a tool that does this automatically. If you’re looking for a fun project, then this might be an interesting thing to investigate!

Scaling Repositories

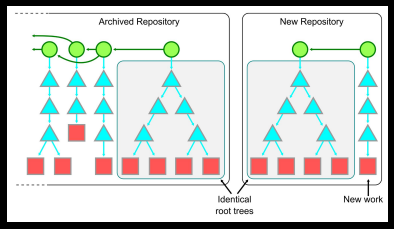
Vertical Sharding (Multi-Repo)



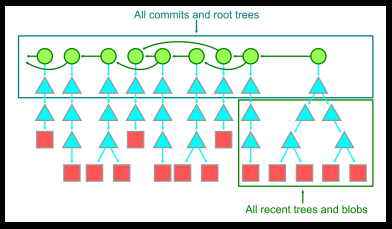
Horizontal Sharding (Submodules)



Time-Based Sharding



Data Offloading



This concludes my interpretation of common database scaling solutions and how they could apply to Git repositories.

Perhaps you have more ideas, and I'd love to hear them.

What can Git learn from other databases?

So this whole time, I've been comparing some critical Git operations as parallels to application databases, and this time focusing on how Git already has some database features.

But, there are so many databases out there and so many more database developers than Git contributors, that there must be some ideas from the world of databases that we have not yet applied to Git. What can Git learn from other databases?

For today, I only want to mention two ideas.

Database Concepts for Git

“Blessed” Sharding Strategies

Git supports submodules as its only officially-supported form of sharding (other than independent multi-repos). Should Git support these other forms more directly?

Alternatively, there is room for third-party tools to create time-based shards or do data offloading without needing to modify the Git client at all. Could be an interesting direction to pursue.

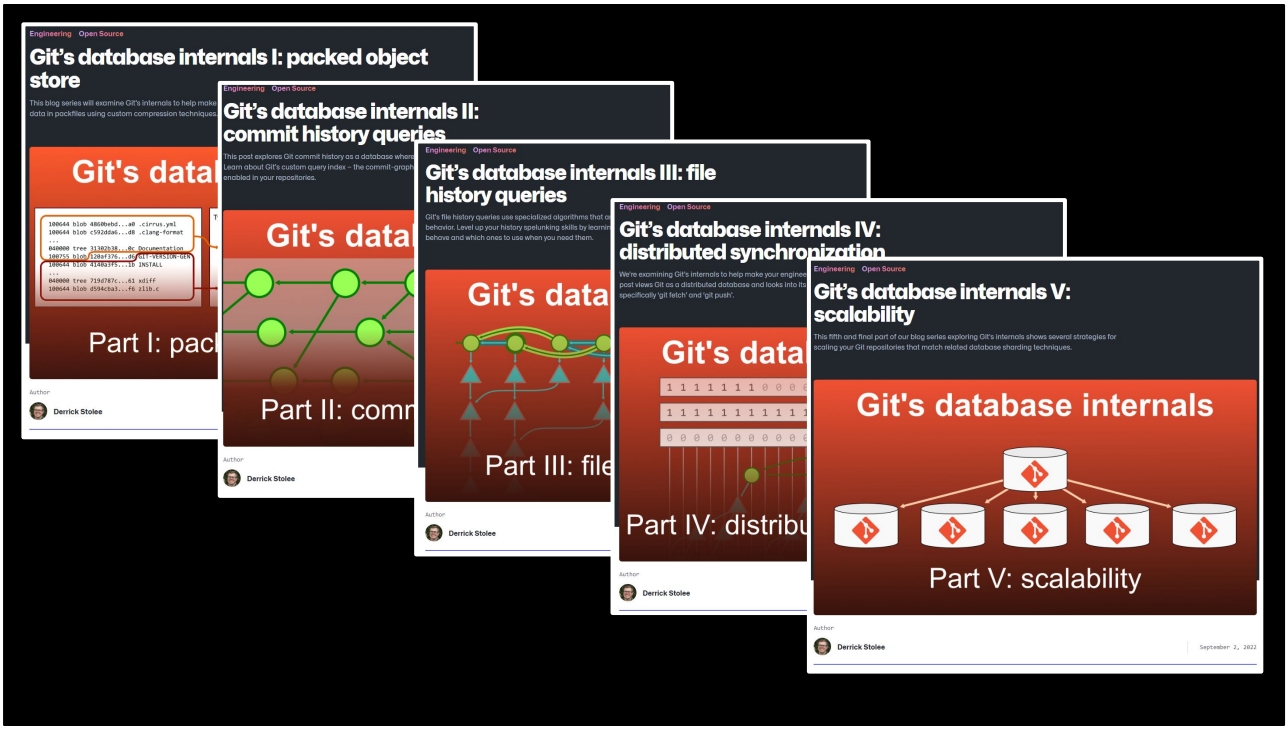
Database Concepts for Git
Dynamic Data Stores

As we discussed earlier today, Git uses the packfile format both for communicating over the network *and* for storing data on-disk.

These pack-files are not mutable after they are written. To repack the object data, Git writes a new packfile and then deletes the old ones. This “rewrite the world” strategy has a difficult time when repositories become very large.

It would be interesting to adapt some database file structures that are built for live-updating and pair them with the delta-compression that exists in pack-files. The important property to keep is that there is low overhead in constructing a pack-file to send over the network while also having maintenance time use resources on the order of the new information to the repository, not the total size of the repository.

This is something I’m particularly interested in and hope to build something soon to share with the community.



Even though my time is limited here on stage, I do want to mention that 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 will take home with you:

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

Thank you!