

DropBox-GoogleDrive

Requirements

- 1) Users can create and upload files
- 2) Files can be shared across multiple different users
- 3) Changes to files should be propagated to all devices with access
- 4) We should be able to see older versions of files

Similar numbers of reads and writes!

Capacity Estimates

- 1) 1 billion users
- 2) ~100 docs per user
- 3) Each doc ~100 kb, some in megabytes
- 4) $1 \text{ billion} \times 100 \times 100 \text{ kb} = 100 \text{ TB}$ of docs
- 5) Most docs shared with < 10 people, some shared with many thousands

Document Permissions

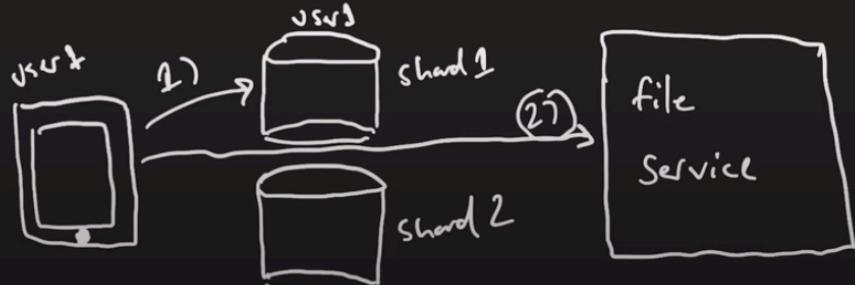
Schema *

User Id	doc Id
1	12
1	13
6	13
6	27

index
and shard!

indexing and partitioning by user Id allows us to quickly see
which documents a given user has access to!

↳ useful on client startup!



Permissions Table

Replication



→ A CRDT could fix this (see add-remove set CRDT)

→ Probably overkill, this table unlikely to be a big bottleneck

→ Let's keep things simple and use MySQL

FILE Uploading

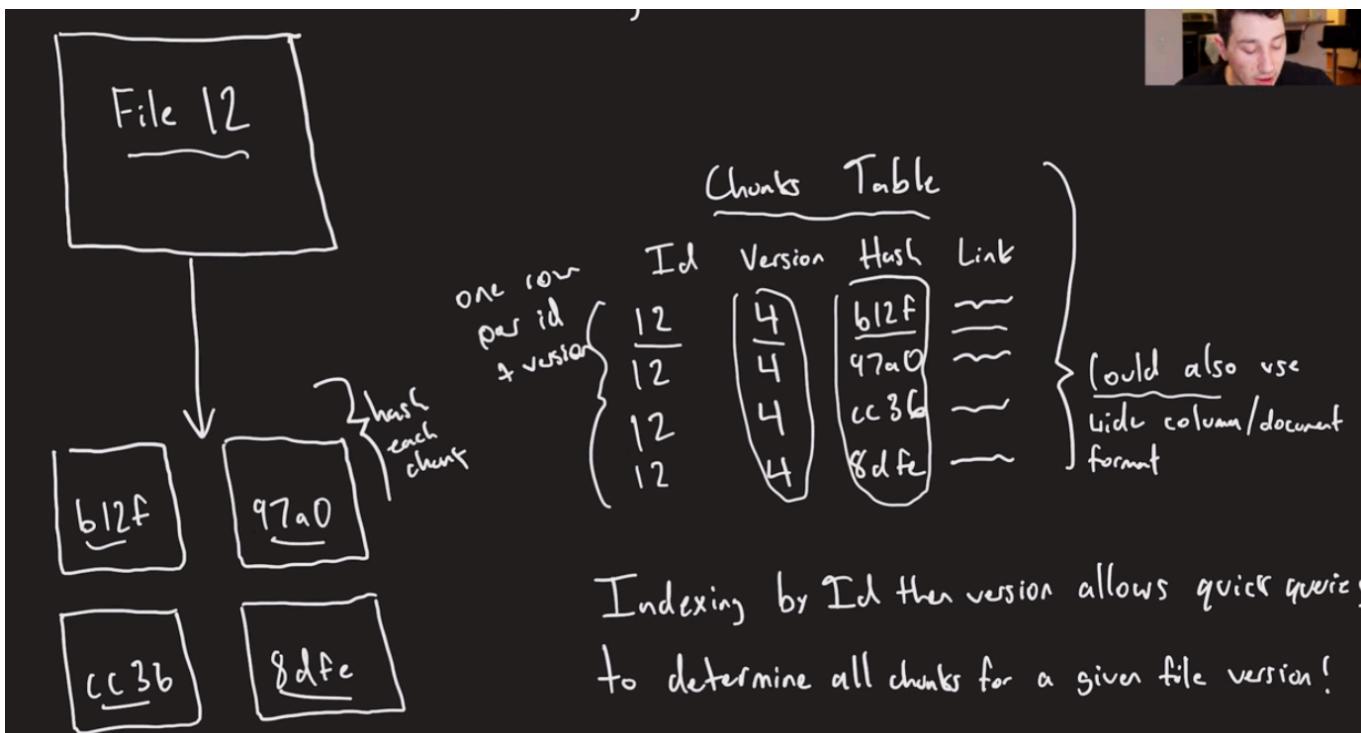


Pros:

- Parallel uploading to S3
- Only load modified chunks on file changes

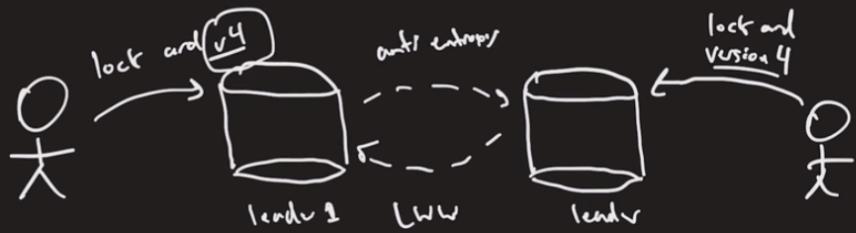
Cons:

- Added complexity on client to do file diffing, have to keep track of chunks in file



Chunks DB consideration

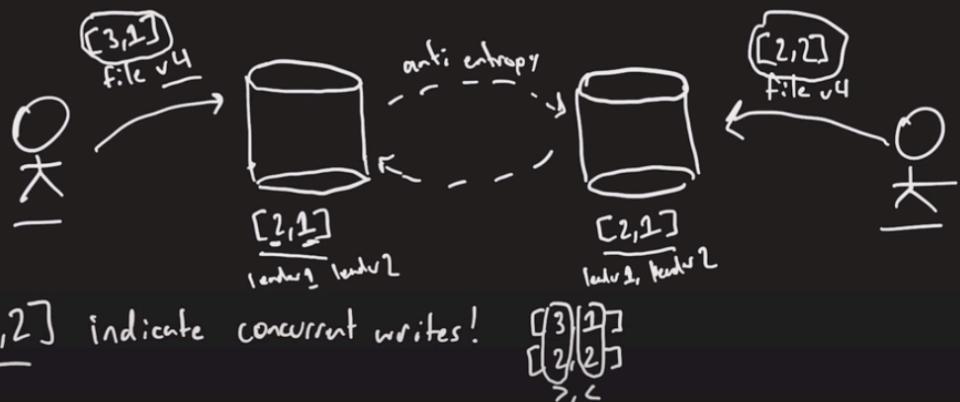
Our locking approach wouldn't work for a multi-leader setup!



Using last write wins, one file version gets clobbered, no history

~~However, if we needed the extra write throughout we could still make this work!~~

Idea: Use version vectors to determine concurrent writes and store both copies!



[3,1] and [2,2] indicate concurrent writes!



Now when we perform anti-entropy, we store file v4.1 and file v4.2

~~Next user to load file sees both versions and must merge locally before upload!~~

Next user to load the file sees both versions and must merge locally.

Lets try to upload using siblings

For the sake of simplicity, let's avoid storing siblings

→ Would be necessary if write throughput needed is super high

→ Could avoid client side merges with CQOT, but we allow storing any filetype

Seems useful to use an ACID database so that all rows are updated atomically

→ Serializability ensures that one version is finished writing before another can publish

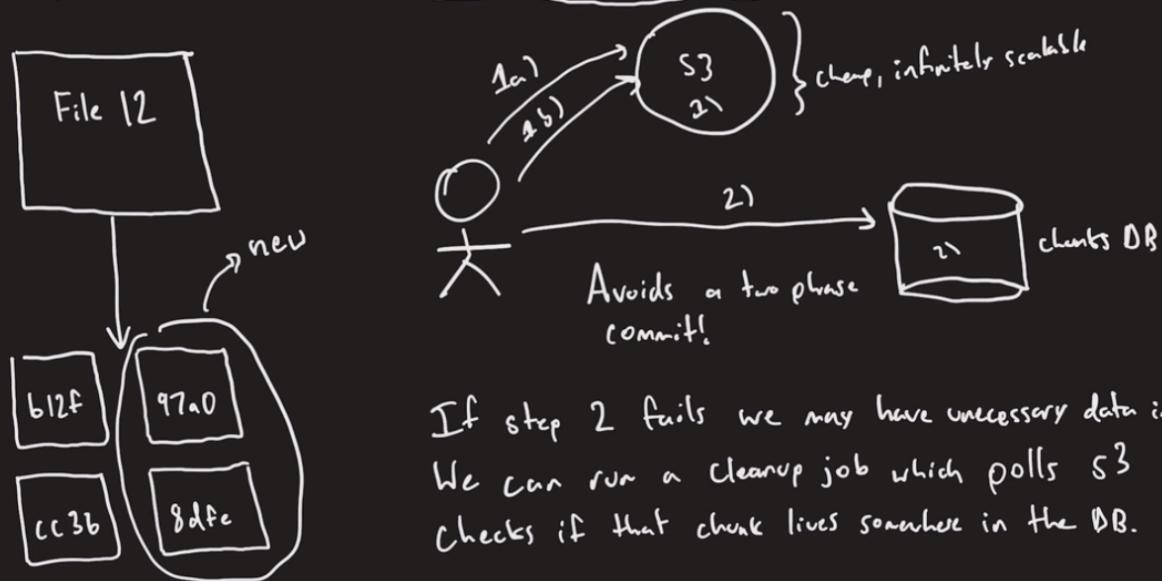
→ If trying to write a version that already exists DB rejects write

Can use MySQL → single leader replication + transactions

→ Increase write throughput by partitioning on fileId

Uploading the actual files

We want to upload as few chunks as possible



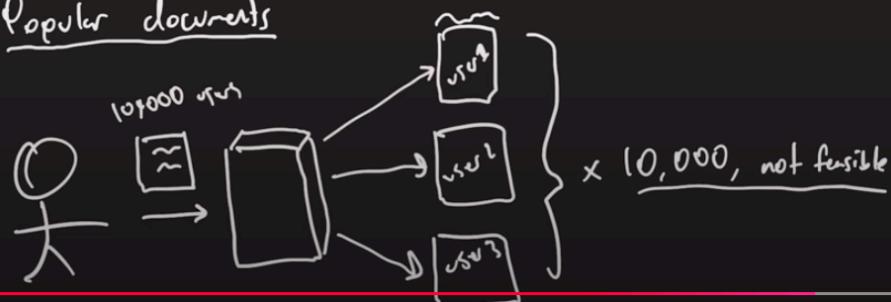
we will upload parallelly the chunks to s3 and when they have completed we will upload to the chunks Db.

Pushing File Changes To Users



We want to ideally keep the number of connections maintained by each client low.
The best way to do this is to have a single node responsible for a given client.
→ Very similar to our news feed design!

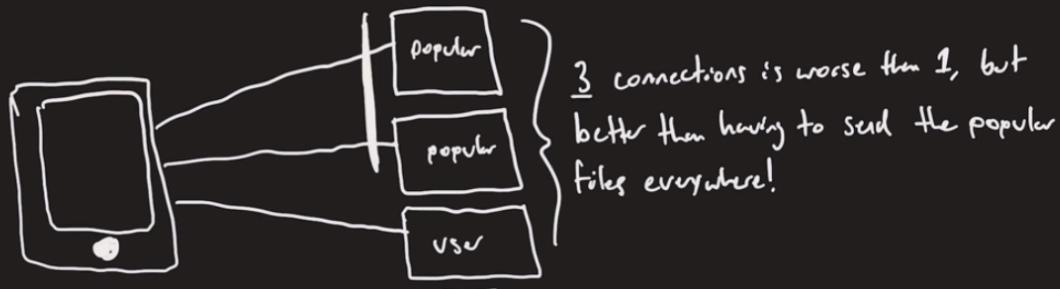
BUT!! Popular documents



Hybrid Approach

For most documents, < 10 users have access
→ We can push these somewhere for the user? shard by user Id

For super popular documents, 100K users have access
→ Push changes to a popular document service ? shard by document Id



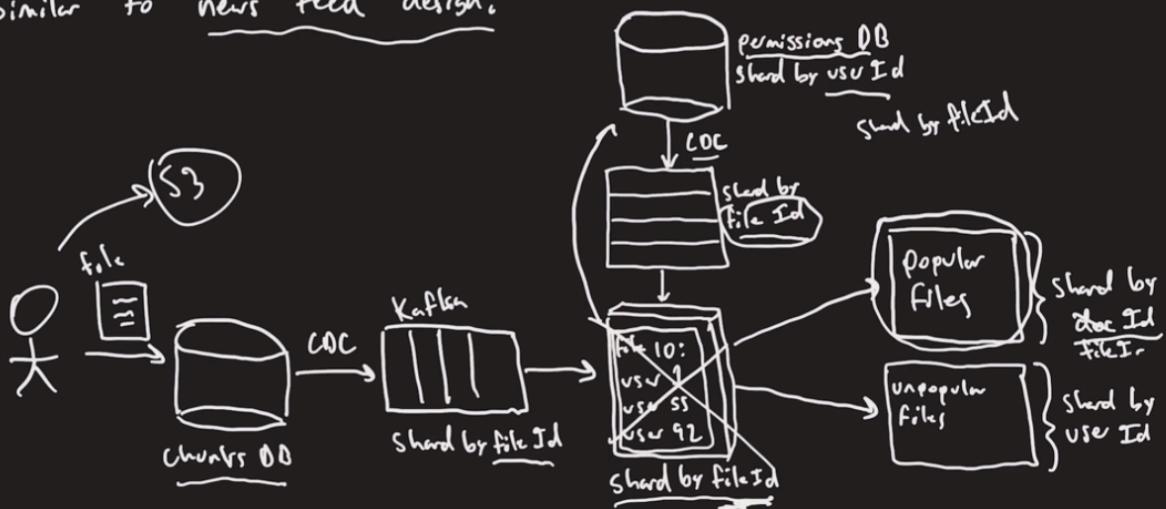
for popular files we will push the new file to a popular document service and the user will open a new connection for that file.

for the unpopular files we will send the file changes to all the users who are using that file.

File Routing

Upon making changes to a file, we need to propagate the changes to

→ Similar to news feed design!



Using Flink ensures that we store a local copy of permissions and don't have to query Db

Using flink ensures that we store a local copy of permissions and dont have to query Db

Metadata REading

Metadata Reading

Since we will have many consumers of each metadata change,
something like Kafka makes sense here (uses long polling under the hood)

→ Persistence ensures that clients that go down can re-read missed changes!

