DESIGN GOOGLE DRIVE

Question 1
Q: Are we just designing the storage aspect of Google Drive, or are we also designing some of the related products like Google Docs, Sheets, Slides, Drawings, etc.?

A: We're just designing the core Google Drive product, which is indeed the storage product. In other words, users can create folders and upload files, which effectively stores them in the cloud. Also, for simplicity, we can refer to folders and files as "entities".

Question 2
Q: There are a lot of features on Google Drive, like shared company drives vs. personal drives, permissions on entities (ACLs), starred files, recently-accessed files, etc.. Are we designing all of these features or just some of them?

A: Let's keep things narrow and imagine that we're designing a personal Google Drive (so you can forget about shared company drives). In a personal Google Drive, users can store entities, and that's all that you should take care of. Ignore any feature that isn't core to the storage aspect of Google Drive; ignore things like starred files, recently-accessed files, etc.. You can even ignore sharing entities for this design.

Question 3
Q: Since we're primarily concerned with storing entities, are we supporting all basic CRUD operations like creating, deleting, renaming, and moving entities?

A: Yes, but to clarify, creating a file is actually uploading a file, folders have to be created (they can't be uploaded), and we also want to support downloading files.

Question 4
Q: Are we just designing the Google Drive web application, or are we also designing a desktop client for Google drive?

A: We're just designing the functionality of the Google Drive web application.

Question 5
Q: Since we're not dealing with sharing entities, should we handle multiple users in a single folder at the same time, or can we assume that this will never happen?

A: While we're not designing the sharing feature, let's still handle what would happen if multiple clients were in a single folder at the same time (two tabs from the same browser, for example). In this case, we would want changes made in that folder to be reflected to all clients within 10 seconds. But for the purpose of this question, let's not worry about conflicts or anything like that (i.e., assume that two clients won't make changes to the same file or folder at the same time).

Question 6
Q: How many people are we building this system for?

A: This system should serve about a billion users and handle 15GB per user on average.

Question 7
Q: What kind of reliability or guarantees does this Google Drive service give to its users?

A: First and foremost, data loss isn't tolerated at all; we need to make sure that once a file is uploaded or a folder is created, it won't disappear until the user deletes it. As for availability, we need this system to be highly available.

1. GATHERING SYSTEM REQUIREMENTS

As with any systems design interview question, the first thing that we want to do is to gather system requirements; we need to figure out what system we're building exactly.

We're designing the core user flow of the **Google Drive** web application. This consists of storing two main entities: folders and files. More specifically, the system should allow users to create folders, upload and download files, and rename and move entities once they're stored. We don't have to worry about ACLs, sharing entities, or any other auxiliary Google Drive features.

We're going to be building this system at a very large scale, assuming 1 billion users, each with **15GB** of data stored in Google Drive on average. This adds up to approximately **15,000 PB** of data in total, without counting any metadata that we might store for each entity, like its name or its type.

We need this service to be **Highly Available** and also very redundant. No data that's successfully stored in Google Drive can ever be lost, even through catastrophic failures in an entire region of the world.

2. COMING UP WITH A PLAN

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, distinguishable components of our how system?

First of all, we'll need to support the following operations:

- For **Files –** *UploadFile, DownloadFile, DeleteFile, RenameFile, MoveFile,*
- For **Folders –** *CreateFolder, GetFolder, DeleteFolder, RenameFolder, MoveFolder*

Secondly, we'll have to come up with a proper storage solution for two types of data:

- **File Contents**: The contents of the files uploaded to Google Drive. These are opaque bytes with no particular structure or format.
- Entity Info: The metadata for each entity. This might include fields like **entityID, ownerID, lastModified, entityName, entityType**. This list is non-exhaustive, and we'll most likely add to it later on.

Let's start by going over the storage solutions that we want to use, and then we'll go through what happens when each of the operations outlined above is performed.

3. STORING ENTITY INFO

To store entity information, we can use **key-value stores**. Since we need high availability and data replication, we need to use something like Etcd, Zookeeper, or Google Cloud Spanner (as a K-V store) that gives us both of those guarantees as well as consistency (as opposed to DynamoDB, for instance, which would give us only eventual consistency).

Since we're going to be dealing with many gigabytes of entity information (given that we're serving a billion users), we'll need to **shard** this data across multiple clusters of these K-V stores. Sharding on entityID means that we'll lose the ability to perform batch operations, which these key-value stores give us out of the box and which we'll need when we move entities around (for instance, moving a file from one folder to another would involve editing the metadata of 3 entities; if they were located in 3 different shards that wouldn't be great). Instead, we can shard based on the **ownerID** of the entity, which means that we can edit the metadata of multiple entities atomically with a transaction, so long as the entities belong to the same user.

Given the traffic that this website needs to serve, we can have a **layer of proxies** for entity information, **load balanced** on a **hash** of the **ownerID**. The proxies could have some **caching**, as well as perform **ACL**(Access Control List – A permission model (list) which specifies what operations can a user perform on the given data.) checks when we eventually decide to support them. The proxies would live at the regional level, whereas the source-of-truth key-value stores would be accessed globally.

4. STORING FILE DATA

When dealing with potentially very large uploads and data storage, it's often advantageous to **split up data into blobs** that can be pieced back together to form the original data. When uploading a file, the request will be load balanced across multiple servers that we'll call "**blob splitters**", and these blob splitters will have the job of splitting files into blobs and storing these blobs in some global blob-storage solution like **GCS** or **S3** (since we're designing **Google** Drive, it might not be a great idea to pick S3 over GCS :P).

One thing to keep in mind is that we need a lot of **redundancy** for the data that we're uploading in order to prevent data loss. So we'll probably want to adopt a strategy like: try pushing to 3 different GCS **buckets** and consider a write successful only if it went through in at least 2 buckets. This way we always have redundancy without necessarily sacrificing availability. In the background, we can have an extra service in charge of further replicating the data to other buckets in an async manner. For our main 3 buckets, we'll want to pick buckets in **3 different availability zones** to avoid having all of our redundant storage get wiped out by potential catastrophic failures in the event of a natural disaster or huge power outtage.

In order to avoid having multiple identical blobs stored in our blob stores, we'll name the blobs after a hash of their content. This technique is called **Content-Addressible Storage**, and by using it, we essentially make all blobs immutable in storage. When a file changes, we simply upload the entire new resulting blobs under their new names computed by hashing their new contents.

This immutability is *very* powerful, in part because it means that we can very easily introduce a caching layer between the blob splitters and the buckets, without worrying about keeping caches in sync with the main source of truth when edits are made--an edit just means that we're dealing with a completely different blob.

5. ENTITY INFO STRUCTURE

Since folders and files will both have common bits of metadata, we can have them share the same structure. The difference will be that folders will have an **is_folder** flag set to true and a list of **children_ids**, which will point to the entity information for the folders and files within the folder in question. Files will have an **is_folder** flag set to false and a **blobs** field, which will have the IDs of all of the blobs that make up the data within the relevant file. Both entities can also have a **parent_id** field, which will point to the entity information of the entity's parent folder. This will help us quickly find parents when moving files and folders.

- File Info

```
{
  blobs: ['blob_content_hash_0', 'blob_content_hash_1'],
  id: 'some_unique_entity_id'
  is_folder: false,
  name: 'some_file_name',
  owner_id: 'id_of_owner',
  parent_id: 'id_of_parent',
}
```

- Folder Info

```
{
  children_ids: ['id_of_child_0', 'id_of_child_1'],
  id: 'some_unique_entity_id'
  is_folder: true,
  name: 'some_folder_name',
  owner_id: 'id_of_owner',
  parent_id: 'id_of_parent',
}
```

6. GARBAGE COLLECTION

Any change to an existing file will create a whole new blob and de-reference the old one. Furthermore, any deleted file will also de-reference the file's blobs. This means that we'll eventually end up with a lot of **orphaned** blobs that are basically unused and taking up storage for no reason. We'll need a way to get rid of these blobs to free some space.

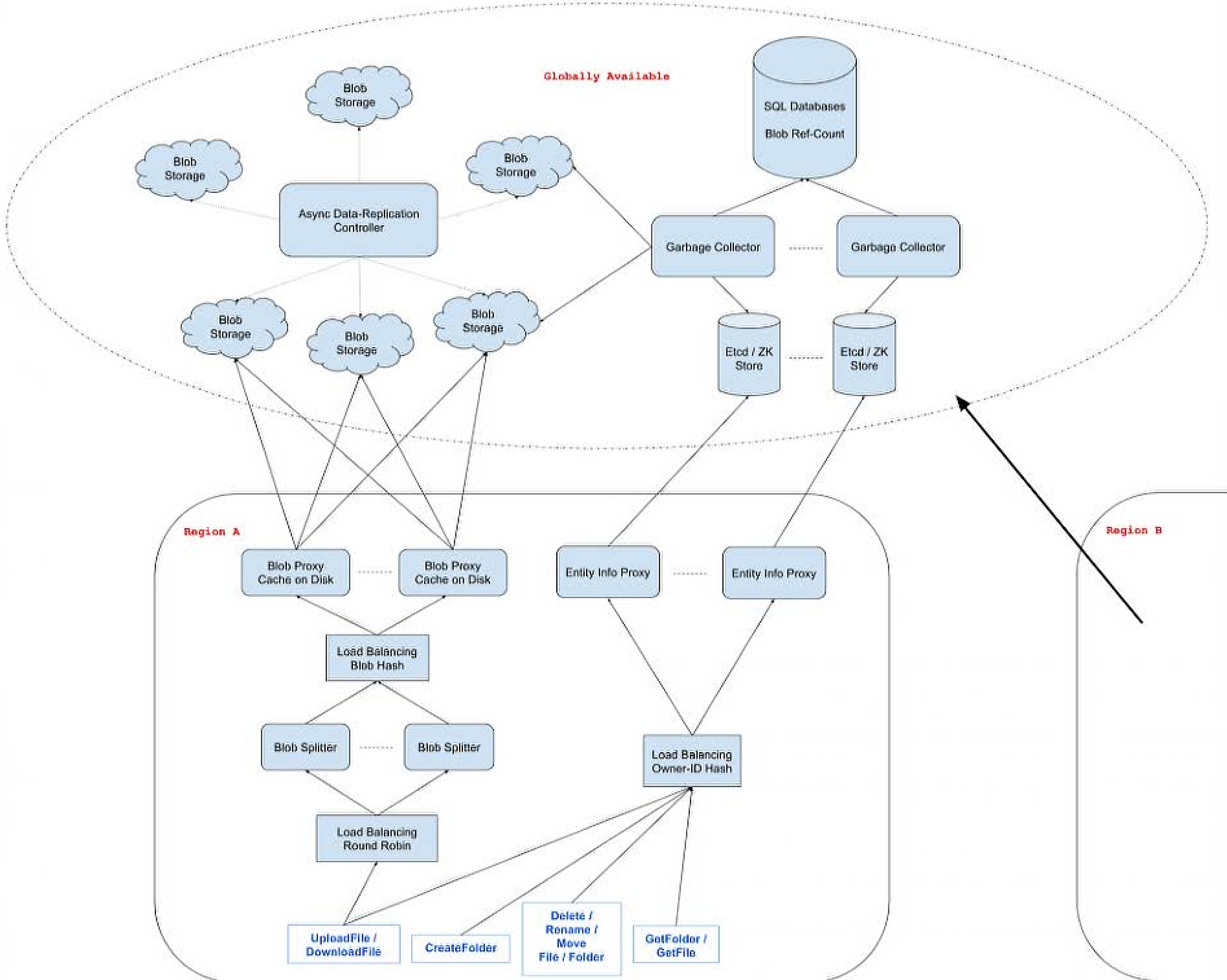
We can have a **Garbage Collection service** that watches the entity-info K-V stores and keeps counts of the number of times every blob is referenced by files; these counts can be stored in a SQL table.

Reference counts will get updated whenever files are uploaded and deleted. When the reference count for a particular blob reaches 0, the Garbage Collector can mark the blob in question as orphaned in the relevant blob stores, and the blob will be safely deleted after some time if it hasn't been accessed.

7. END TO END API FLOW

Now that we've designed the entire system, we can walk through what happens when a user performs any of the operations we listed above.

- ***CreateFolder*** - is simple; since folders don't have a blob-storage component, creating a folder just involves storing some metadata in our key-value stores.
- ***UploadFile*** - works in two steps. The first is to store the blobs that make up the file in the blob storage. Once the blobs are persisted, we can create the file-info object, store the blob-content hashes inside its **blobs** field, and write this metadata to our key-value stores.
- Because we have the Content Addressable Storage system set up, even when an upload file operation fails and then starts over the next time, we won't be duplicating the stored data as this system won't ever allow us to duplicate data.
- ***DownloadFile*** - fetches the file's metadata from our key-value stores given the file's ID. The metadata contains the hashes of all of the blobs that make up the content of the file, which we can use to fetch all of the blobs from blob storage. We can then assemble them into the file and save it onto local disk.
- ***RenameFolder*** – Just go to k-v store and edit the name property for this particular folder.
- ***Move*** – Update the "parent" property in k-v store for folder we're moving and the "child" property for both the current parent folder and the destinations parent folder in the k-v store.
- Each folder is like a tree data structure with a pointer to its parent i.e. a parent property in its k-v store with value set to parentId
- ***Get*** - Achieved by polling every 10 sec from the client. Also, here we can use the proxies to k-v stores. Whenever an entity info changes, set the last modified time for it and these proxies can figure out if something has changed recently or not and accordingly, they push last modified time and updated data to client.


Globally Available
Blob Storage
Blob Storage
Blob Storage
Async Data-Replication Controller
SQL Databases
Blob Ref-Count
Garbage Collector
Garbage Collector
Blob Storage
Blob Storage
Blob Storage
Etcd / ZK Store
Etcd / ZK Store
Region A
Region B
Blob Proxy Cache on Disk
Blob Proxy Cache on Disk
Entity Info Proxy
Entity Info Proxy
Load Balancing Blob Hash
Blob Splitter
Blob Splitter
Load Balancing Owner-ID Hash
Load Balancing Round Robin
UploadFile / DownloadFile
CreateFolder
Delete / Rename / Move File / Folder
GetFolder / GetFile