Capacity Estimation

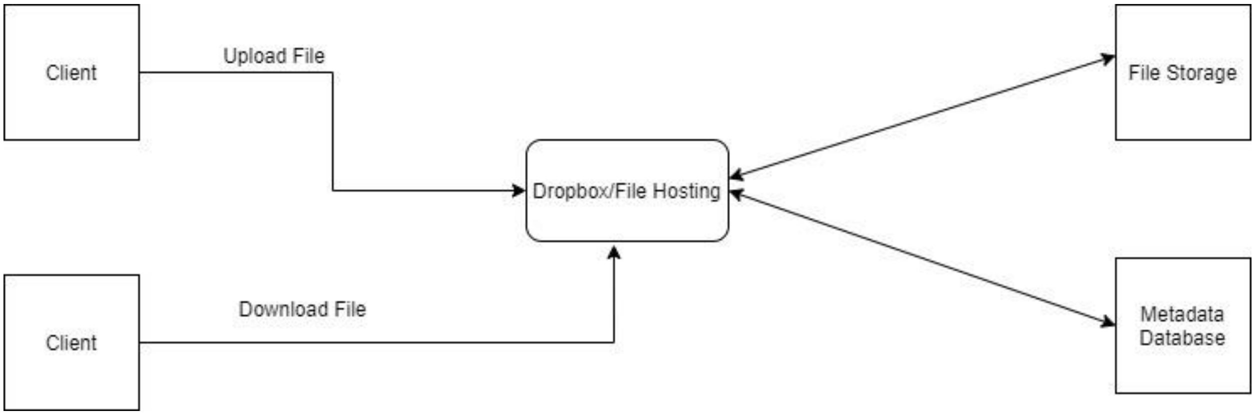
- Let’s assume that we have 500M total users, and 100M daily active users (DAU). Let’s assume that on average each user connects from three different devices. On average if a user has 200 files/photos, we will have 100 billion total files
- Let’s assume that average file size is 100KB, this would give us ten petabytes of total storage.

  100B * 100KB => 10PB
- Let’s also assume that we will have one million active connections per minute.

Constraints

We should expect huge read and write volumes. Read to write ratio is expected to be nearly the same. Internally, files can be stored in small parts or chunks (say 4MB); this can provide a lot of benefits i.e. all failed operations shall only be retried for smaller parts of a file. If a user fails to upload a file, then only the failing chunk will be retried. We can reduce the amount of data exchange by transferring updated chunks only. By removing duplicate chunks, we can save storage space and bandwidth usage. Keeping a local copy of the metadata (file name, size, etc.) with the client can save us a lot of round trips to the server. For small changes, clients can intelligently upload the diffs instead of the whole chunk.

High Level Design



System APIs

Upload file(User_id, filename, file id)

User_id → id of user uploading file/photos/videos

Filename →file to be uploaded

File id → unique id of file to be uploaded

Download file (User_id ,file id, filename)

User_id → id of user uploading file/photos/videos

Filename →file to be uploaded

File id → unique id of file to be uploaded

Database Design

- System is both read and write heavy
- Avg file size is 10 mb

User

User_id	Number
Username	String
Password	String
Quota Assigned	String
Quote Used	String
Created date	Datetime
Last login date	Datetime

Metadata Table

Chunk Id	String
Chunk order	Number
Object version	Number
Is Folder	Boolean
Object modified	Number

Filename	String
File extension	String
File size	Number
File path	String
Creation date	Datetime
Last modified date	Datetime

User Files

User_id	Number
Chunk id	String

A straightforward approach to store above table in relational table such as MySQL. For better consistency, scalability, and availability, we propose to utilize

We can store files in a distributed file storage like HDFS or S3.

We need to store relationships between users and metadata to know who owns which files. For this, we can use a wide column datastore like Cassandra. For the 'User ' table, the 'key' would be 'User_id' and the 'value' would be the list of 'Chunk IDs' the user owns, stored in different columns.

Cassandra or key-value stores in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes do not get applied instantly, data is retained for certain days (to support undeleting) before getting removed from the system permanently

Basic System Design and Algorithm

Let us go through the major components of our system one by one:

a. Client

The Client Application monitors the workspace folder on user’s machine and syncs all files/folders in it with the remote Cloud Storage. The client application will work with the storage servers to upload, download, and modify actual files to backend Cloud Storage. The client also interacts with the remote Synchronization Service to handle any file metadata updates e.g. change in the file name, size, modification date, etc.

Here are some of the essential operations of the client:

1. Upload and download files.
2. Detect file changes in the workspace folder.
3. Handle conflict due to offline or concurrent updates.

How do we handle file transfer efficiently? As mentioned above, we can break each file into smaller chunks so that we transfer only those chunks that are modified and not the whole file. Let us say we divide each file into fixed size of 4MB chunks. We can statically calculate what could be an optimal chunk size based on 1) Storage devices we use in the cloud to optimize space utilization and Input/output operations per second (IOPS) 2) Network bandwidth 3) Average file size in the storage etc. In our metadata, we should also keep a record of each file and the chunks that constitute it.

Should we keep a copy of metadata with Client? Keeping a local copy of metadata not only enable us to do offline updates but also saves a lot of round trips to update remote metadata.

How can clients efficiently listen to changes happening on other clients? One solution could be that the clients periodically check with the server if there are any changes. The problem with this approach is that we will have a delay in reflecting changes locally as clients will be checking for changes periodically compared to server notifying whenever there is some change. If the client frequently checks the server for changes, it will not only be wasting bandwidth, as the server must return empty response most of the time but will also be keeping the server busy. Pulling information in this manner is not scalable too.

A solution to above problem could be to use HTTP long polling. With long polling, the client requests information from the server with the expectation that the server may not respond immediately. If the server has no new data for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available. Once it does have new information, the server immediately sends an HTTP/S response to the client, completing the open HTTP/S Request. Upon receipt of the server response, the client can immediately issue another server request for future updates.

Based on the above considerations we can divide our client into following four parts:

I. Internal Metadata Database will keep track of all the files, chunks, their versions, and their location in the file system.

II. Chunker will split the files into smaller pieces called chunks. It will also be responsible for reconstructing a file from its chunks. Our chunking algorithm will detect the parts of the files that have been modified by the user and only transfer those parts to the Cloud Storage; this will save us bandwidth and synchronization time.

III. Watcher will monitor the local workspace folders and notify the Indexer of any action performed by the users, e.g., when users create, delete, or update files or folders. Watcher also listens to any changes happening on other clients that are broadcasted by Synchronization service.

IV. Indexer will process the events received from the Watcher and update the internal metadata database with information about the chunks of the modified files. Once the chunks are successfully submitted/downloaded to the Cloud Storage, the Indexer will communicate with the remote Synchronization Service to broadcast changes to other clients and update remote metadata database.

How should clients handle slow servers? Clients should exponentially back-off if the server is busy/not-responding. Meaning, if a server is too slow to respond, clients should delay their retries, and this delay should increase exponentially.

Should mobile clients sync remote changes immediately? Unlike desktop or web clients, that check for file changes on a regular basis, mobile clients usually sync on demand to save user's bandwidth and space.

b. Metadata Database

The Metadata Database is responsible for maintaining the versioning and metadata information about files/chunks, users, and workspaces. The Metadata Database can be a relational database such as MySQL, or a NoSQL database service such as DynamoDB. Regardless of the type of the database, the Synchronization Service should be able to provide a consistent view of the files using a database, especially if more than one user work with the same file simultaneously. Since NoSQL data stores do not support ACID properties in favor of scalability and performance, we need to incorporate the support for ACID properties programmatically in the logic of our Synchronization Service in case we opt for this kind of databases. However, using a relational database can simplify the implementation of the Synchronization Service as they natively support ACID properties.

Metadata Database should be storing information about following objects:

1. Chunks
2. Files
3. User
4. Devices
5. Workspace (sync folders)

c. Synchronization Service

The Synchronization Service is the component that processes file updates made by a client and applies these changes to other subscribed clients. It also synchronizes clients' local databases with the information stored in the remote Metadata DB. The Synchronization Service is the most important part of the system architecture due to its critical role in managing the metadata and synchronizing users' files. Desktop clients communicate with the Synchronization Service to either obtain updates from the Cloud Storage or send files and updates to the Cloud Storage and potentially other users. If a client was offline for a period, it polls the system for new updates as soon as it becomes online. When the Synchronization Service receives an update request, it checks with the Metadata Database for consistency and then proceeds with the update. Subsequently, a notification is sent to all subscribed users or devices to report the file update.

The Synchronization Service should be designed in such a way to transmit less data between clients and the Cloud Storage to achieve better response time. To meet this design goal, the Synchronization Service can employ a differencing algorithm to reduce the amount of the data that needs to be synchronized. Instead of transmitting entire files from clients to the server or vice versa, we can just transmit the difference between two versions of a file. Therefore, only the part of the file that has been changed is transmitted. This also decreases bandwidth consumption and cloud data storage for the end user. As described above we will be dividing our files into 4MB chunks and will be transferring modified chunks only. Server and clients can calculate a hash (e.g., SHA-256) to see whether to update the local copy of a chunk or not. On server if we already have a chunk with a similar hash (even from another user) we do not need to create another copy, we can use the same chunk.

To be able to provide an efficient and scalable synchronization protocol we can consider using a communication middleware between clients and the Synchronization Service. The messaging middleware should provide scalable message queuing and change notification to support a high number of clients using pull or push strategies. This way, multiple Synchronization Service instances can receive requests from a global request Queue, and the communication middleware will be able to balance their load.

d. Message Queuing Service

An important part of our architecture is a messaging middleware that should be able to handle a substantial number of requests. A scalable Message Queuing Service that supports asynchronous message-based communication between clients and the Synchronization Service instances best fits the requirements of our application. The Message Queuing Service supports asynchronous and loosely coupled message-based communication between distributed components of the system. The Message Queuing Service should be able to efficiently store any number of messages in a highly available, reliable, and scalable queue.

Message Queuing Service will implement two types of queues in our system. The Request Queue is a global queue, and all client will share it. Clients' requests to update the Metadata Database will be sent to the Request Queue first, from there Synchronization Service will take it to update metadata. The Response Queues that correspond to individual subscribed clients are responsible for delivering the update messages to each client. Since a message will be deleted from the queue once received by a client, we need to create separate Response Queues for each subscribed client to share update messages.

e. Cloud/Block Storage

Cloud/Block Storage stores chunks of files uploaded by the users. Clients directly interact with the storage to send and receive objects from it. Separation of the metadata from storage enables us to use any storage either in cloud or in-house.

Data Partitioning and Replication

To scale out metadata DB, we need to partition it so that it can store information about millions of users and billions of files/chunks. We need to come up with a partitioning scheme that would divide and store our data to different DB servers.

1. Vertical Partitioning: We can partition our database in such a way that we store tables related to one feature on one server. For example, we can store all the user related tables in one database and all files/chunks related tables in another database. Although this approach is straightforward to implement it has some issues:

    1. Will we still have scale issues? What if we have trillions of chunks to be stored and our database cannot support to store such huge number of records? How would we further partition such tables?
    2. Joining two tables in two separate databases can cause performance and consistency issues. How frequently do we have to join user and file tables?

2. Range Based Partitioning: What if we store files/chunks in separate partitions based on the first letter of the File Path. So, we save all the files starting with letter 'A' in one partition and

those that start with letter ‘B’ into another partition and so on. This approach is called range-based partitioning. We can even combine certain less frequently occurring letters into one database partition. We should come up with this partitioning scheme statically so that we can always store/find a file in a predictable manner.

The main problem with this approach is that it can lead to unbalanced servers. For example, if we decide to put all files starting with letter ‘E’ into a DB partition, and later we realize that we have too many files that start with letter ‘E’, to such an extent that we cannot fit them into one DB partition.

3. Hash-Based Partitioning: In this scheme we take a hash of the object we are storing and based on this hash we figure out the DB partition to which this object should go. In our case, we can take the hash of the ‘File ID’ of the File object we are storing to determine the partition the file will be stored. Our hashing function will randomly distribute objects into different partitions, e.g., our hashing function can always map any ID to a number between [1…256], and this number would be the partition we will store our object.

This approach can still lead to overloaded partitions, which can be solved by using Consistent Hashing.

Caching

We can have two types of cache in our system.

Hot Files/Chunks Cache

To server hot files/chunks, we can introduce block storage cache, we can use Mem Cache which can store whole chunks with their respective hashes/ids before hitting block server

When the cache is full, and we want to replace a chunk with a newer/hotter chunk, how would we choose? Least Recently Used (LRU) can be a reasonable policy for our system. Under this policy, we discard the least recently used chunk first.

Metadata Cache

We can use Redis Cache for metadata database

Load Balancing

We can add Load balancing layer at two places in our system 1) Between Clients and Block servers and 2) Between Clients and Metadata servers. Initially, a simple Round Robin approach can be adopted; that distributes incoming requests equally among backend servers. This LB is simple to implement and does not introduce any overhead. Another benefit of this approach is if a server is dead, LB will take it out of the rotation and will stop sending any traffic to it. A problem with Round Robin LB is, it will not take server load into consideration. If a server is overloaded or slow, the LB will not stop sending new requests to that server. To handle this, a more intelligent LB solution can be placed that periodically queries backend server about their load and adjusts traffic based on that.