Design Dropbox

15 March 2024

Google drive / onedrive

Friday, March 15, 2024 9:07 PM

FUNCTIONAL REQUIREMENTS

1) upload / download files
2) share files / folders with other users
3) support offline editing
4) automatic synchronization between various devices which we'll see

NON FUNCTIONAL REQUIREMENTS

1) consistency → File consistency — consistency over availability
   → Server consistency — availability over consistency

[Read and Write heavy]

2) ACID properties — for file we need it for sure
   - atomicity, consistency, isolation, durability

3) Reliability — none of the files must be lost
4) Scalability is a given option I consider

CAPACITY ESTIMATION

Assumption 1: 500M total users — 3 devices on an average
2: 100M active users

Traffic 500M × 200 Files = 100 Billion
why are we not using active users? it is difficult to estimate in daily basis

Storage 100 KB × 100 Billion = 10^{16} = 10 PB

Bandwidth

3 devices × 100M × 100 KB (size) × 2 files/day.

= 6 × 100 × 10^6 × 100 × KB

= 60 TB/day

= $\frac{60}{24 \times 60 \times 60}$ TB = (min kal)

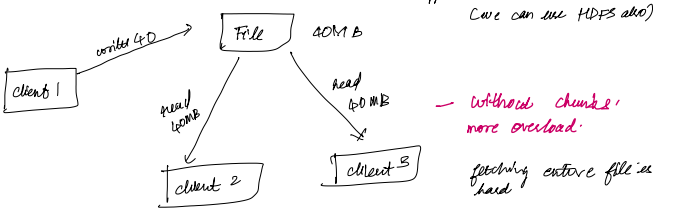
Cache:

200 GB — assumption

DATA MODELLING

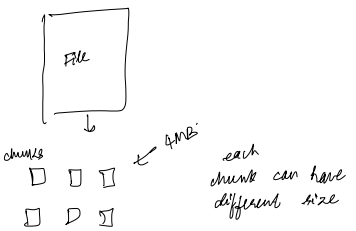
File — File data
File meta data

File System:
Amazon
EBS — supports file hierarchy. (we can use HDFS also)
S3 (object based storage)



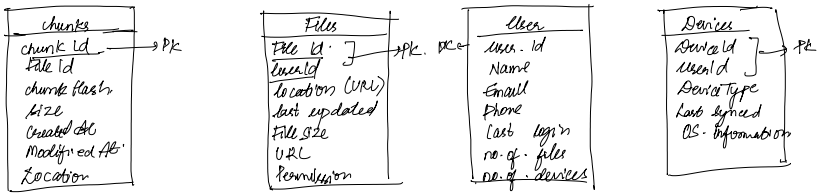
— without chunks, more overload.

fetching entire file is hard



each chunks can have different size

send the updated chunks to the clients



what is workspace

Workspace
Device Id
size
last edited

MOCK INTERVIEW
- GIVE IT A LOOK.

Cache eviction
Load balancing

E
NF
CG
DM
API
High Level design.

SQL vs NO SQL

ACID

NO SQL
cassandra

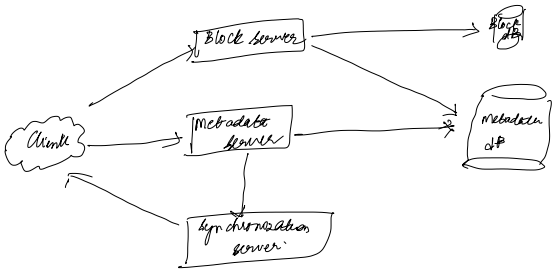
Scalability ✓
ACID ×
+
ACID
in application logic

combination:
SQL + NOSQL
scale → availability
order → consistency

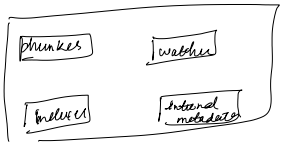
APIs

1. FileId = uploadFile (userId, deviceId, permissions, FileLocation, createdAt,
2. File = downloadFile (userId, fileId, directedFile)
3. user/feature + sharedFile (fileId, list of userIds, permissions, fileLocation)
4. List of = syncDevice (deviceId, userId, lastSynced files/metadata
5. user/feature = updateFile (fileId, userId, chunkHash, fileLocation, modified time, chunk(d)

upload a file goes to blockdb



CLIENT SIDE COMPONENTS

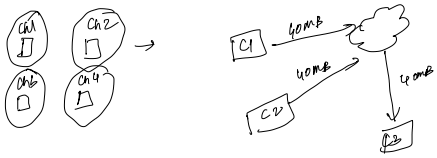


Chunker
divides file and merges it

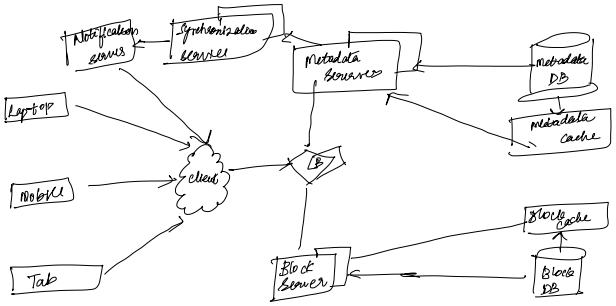
Watcher
checks for any update
look for notifications

Indexer
responsible for synchronization
inform server

Internal metadata
chunk local metadata



chunk size is dependant on
1. average file size
2. network bandwidth
3. server capabilities (IOPS)



LL Design.

Cache Eviction Technique.
LRU

Load Balancing. → can give combination
1. Least response time.
2. Less bandwidth

Partitioning Technique.

range based partitioning

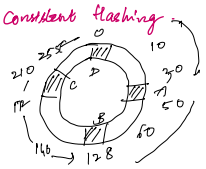
hash(File Id) % 4

can be

0 — 0-1000

1 — 1000 — 2000

Consistent Hashing.



hashing based algorithm which gives answer by only 0 to 255

servers would increase

2 — 2000 - 3000

If server C went down,

A will take it up and handle requests of C

Overburden on A

new server D is added range of A gets disturbed

Introduce 4 — 3

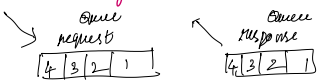
11%04 == 3 11%05 == 1

use replica of A - double hashing

to distribute among the replicas

Questions: How to add extra replicas
security
orchestration of containers

How is synchronisation service get the work done?



Individual response queue for each and every client.

request, response are priority queue

RabbitMQ or Kafka

no displacements if needed

PULL → wastage of bandwidth

PUSH → HTTP long pooling