Hello

CS 744: GOOGLE FILE SYSTEM

Shivaram Venkataraman

Fall 2022

ANNOUNCEMENTS

- Assignment 1 out later today
- Group submission form → 5pm today
- Anybody on the waitlist?

OUTLINE

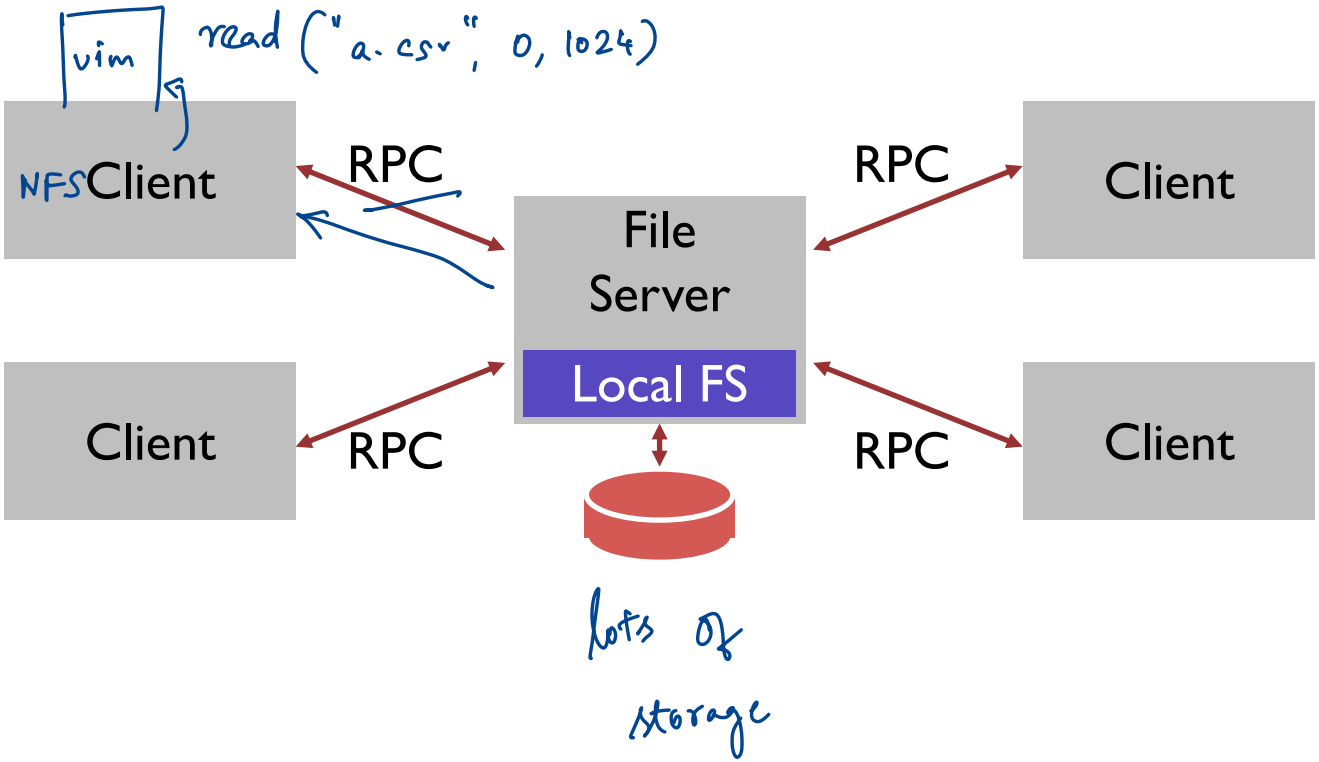
1. Brief history
2. GFS
3. Discussion
4. What happened next?

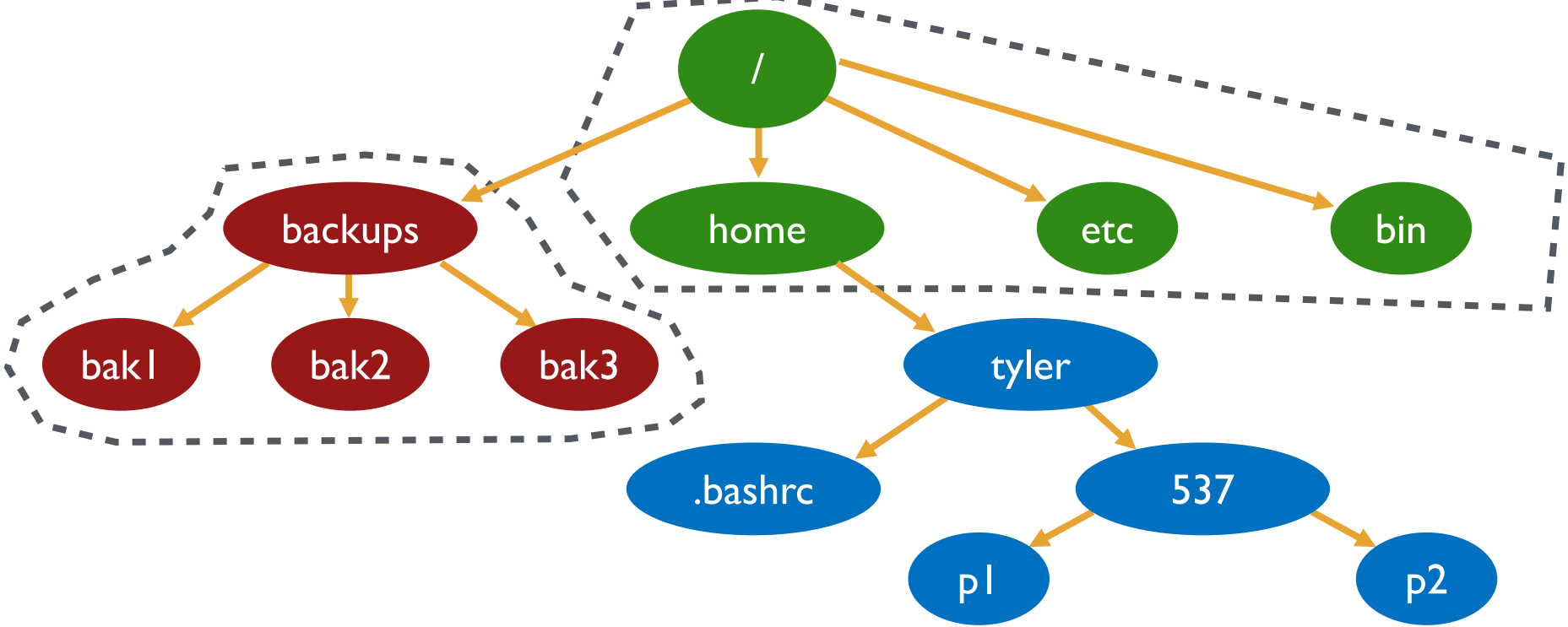
HISTORY OF DISTRIBUTED FILE SYSTEMS

transparent

SUN NFS

mid 1980s





/dev/sda1 **on** /

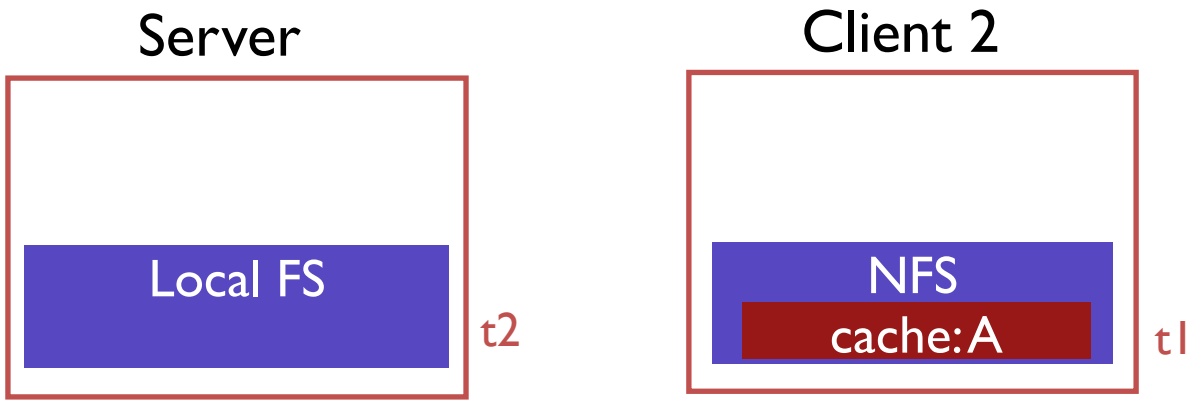
/dev/sdb1 **on** /backups

NFS **on** /home

Caching was popular to improve perf

Caching protocols were non-trivial.

CACHING



Client cache records time when data block was fetched (t1)

Before using data block, client does a STAT request to server

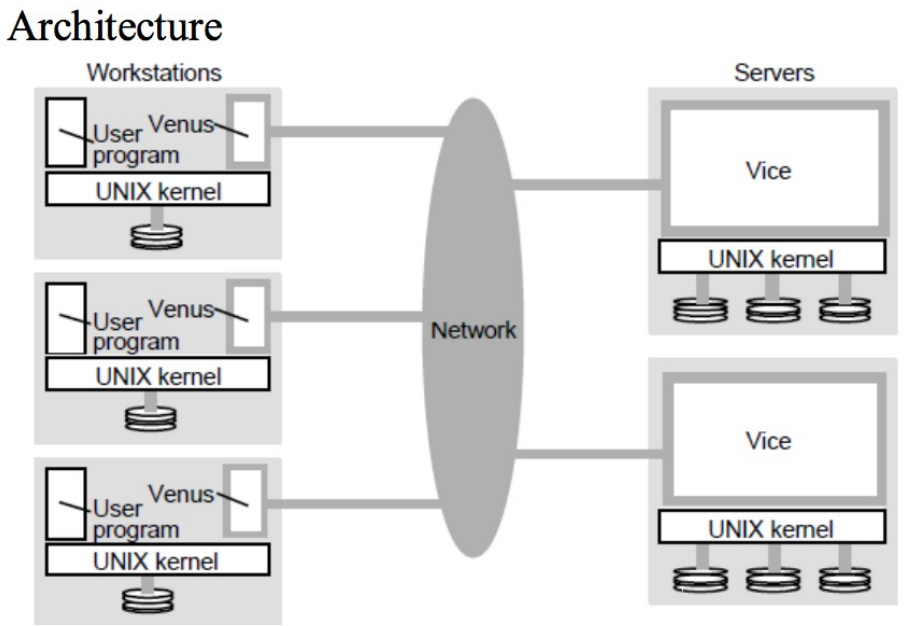
- get's last modified timestamp for this file (t2) (not block…)
- compare to cache timestamp
- refetch data block if changed since timestamp (t2 > t1)

ANDREW FILE SYSTEM → AFS

- Design for scale → larger number of clients

- Whole-file caching
  ↳ client opens a file

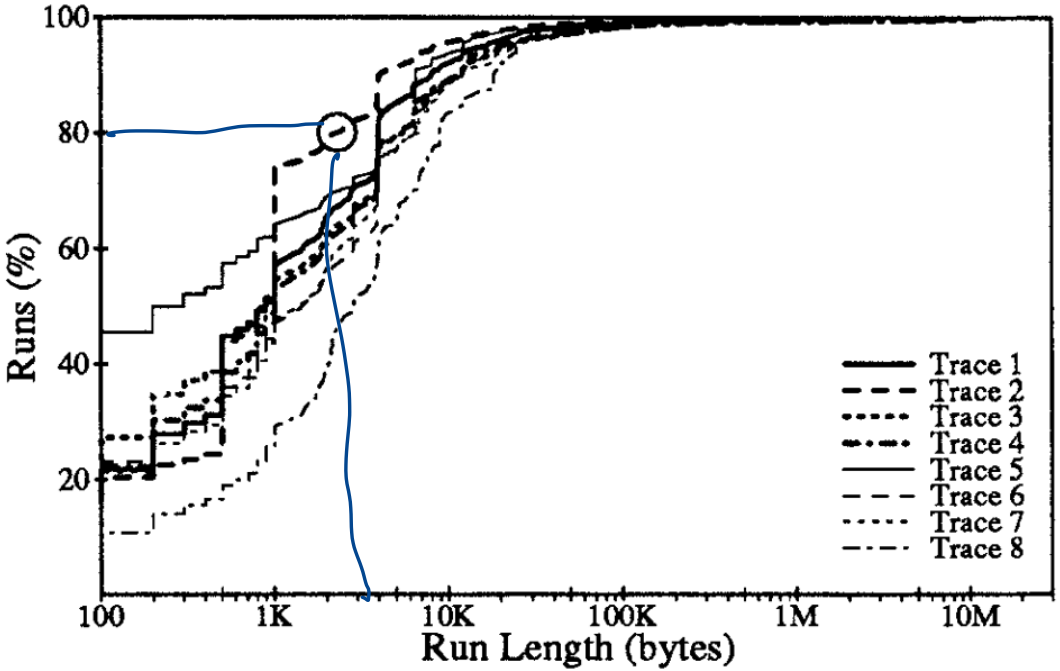
- Callbacks from server



WORKLOAD PATTERNS (1991)

bytes read

< 4K in size



Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout

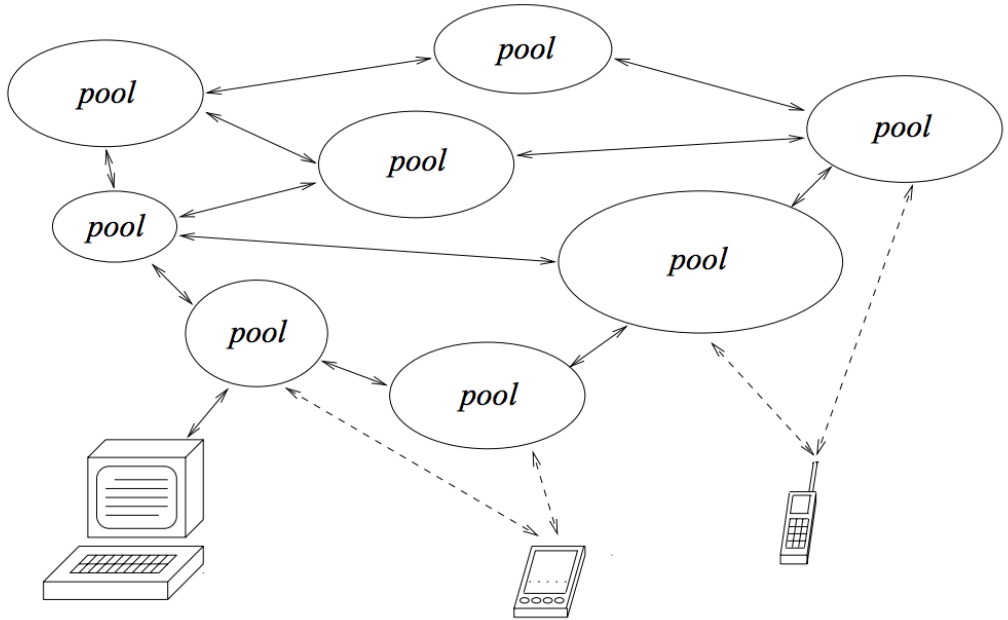
OCEANSTORE/PAST – late 90s early 2000s

Bit Torrent

Wide area storage systems

Fully decentralized

Built on distributed hash tables (DHT)



GFS: WHY ?

Fault tolerance

→ Hardware characteristics

"Large" Sequential reads / writes

very few random reads

~ no random writes

↓

Applications could work without POSIX !

Single master

Centralization

- single enterprise owns the distributed system →

Components with failures

Files are huge !

GFS: WHY ?

Applications are different

GFS: WORKLOAD ASSUMPTIONS

"Modest" number of large files

Two kinds of reads: Large Streaming and small random

Writes: Many large, sequential writes. Few random

High bandwidth more important than low latency

GFS: DESIGN

- Single Master for metadata

- Chunkservers for storing data

- No POSIX API !
- No Caches!

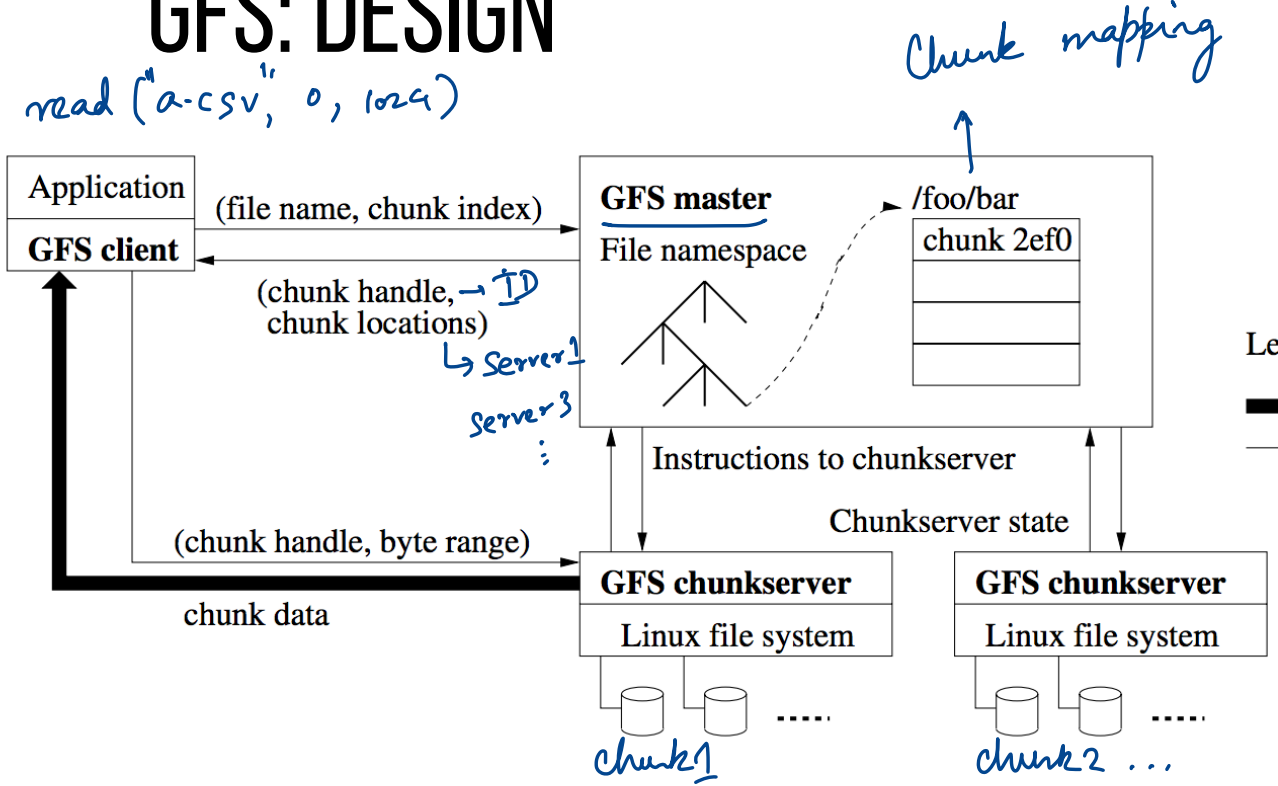


Figure 1: GFS Architecture

Every file divided into chunks

CHUNK SIZE TRADE-OFFS

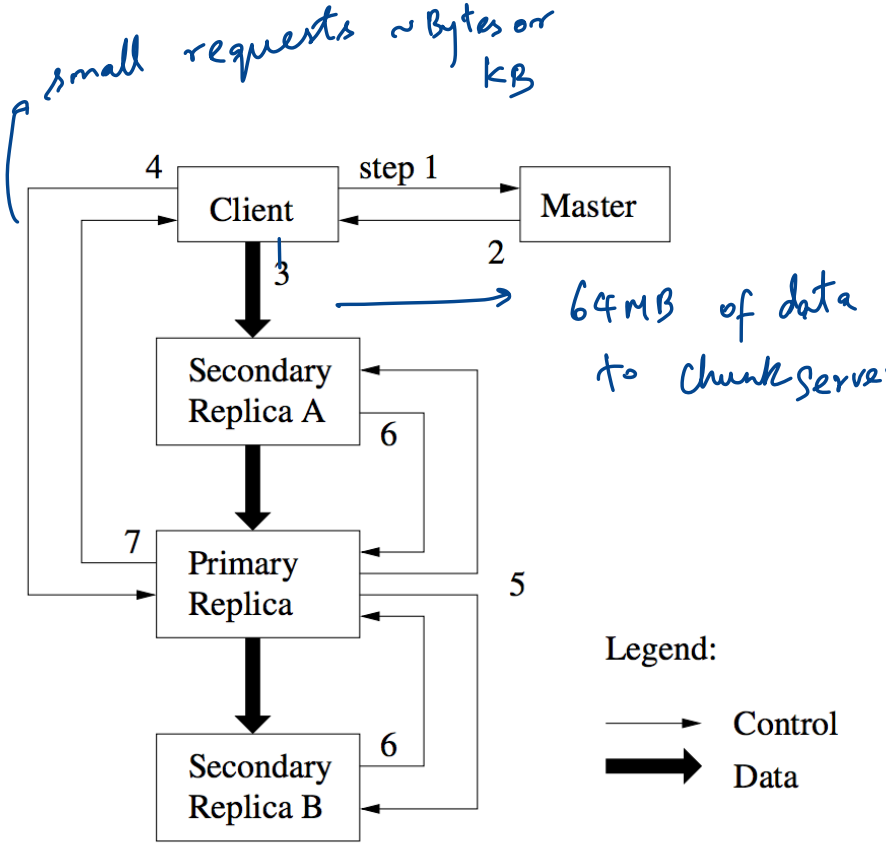
Client → Master ⟶ too small chunk size → too many RPCs to master

Client → Chunkserver

Metadata ⟶ too much metadata if chunks are too small!

Having too large a chunksize
↳ overload a chunk server (fault recovery times)

GFS: REPLICATION



configurable

- 3-way replication to handle faults
- Primary replica for each chunk
- Chain replication (consistency)

- Decouple data, control flow
- Dataflow: Pipelining, network-aware

RECORD APPENDS

Write	Client specifies the offset
Record Append	GFS chooses offset

Web servers produce logs
→ Queries made to Search
→ Entry
["CS744", "Madison", "OK"]

Consistency → Slightly weaker guarantees
- At-least once → It might appear more than once, but it will appear at least once
- Atomic
  - ↳ entire record will appear together

MASTER OPERATIONS

- No "directory" inode! Simplifies locking
- Replica placement considerations
- Implementing deletes

No symlink

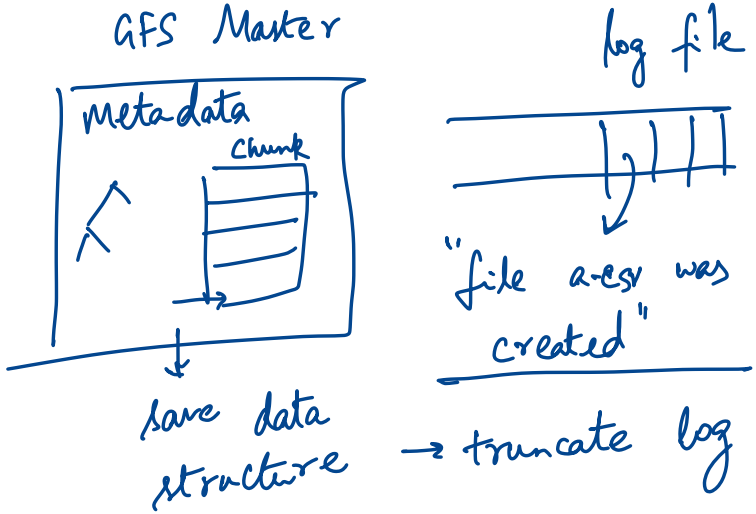
→ failure risk. One replica outside this rack

→ load, disk utilization

→ lazily. Metadata makes a note that file is deleted

FAULT TOLERANCE

- Chunk replication with 3 replicas
- Master
  - Replication of log, checkpoint
  - Shadow master



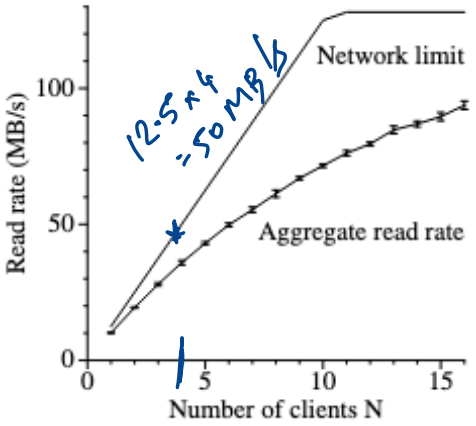
- Data integrity using checksum blocks

chunkserver encode checksums ~32 KB | 64 KB

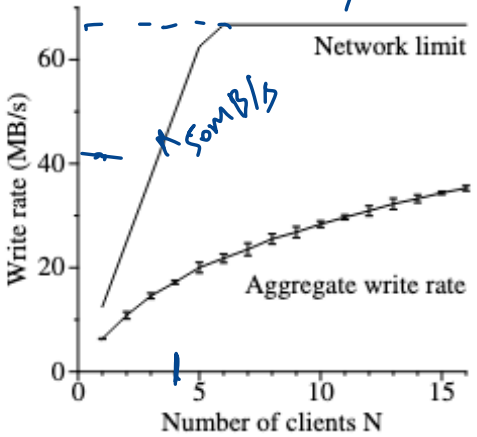
DISCUSSION

https://forms.gle/DntYB3yTL9eQZFzKA

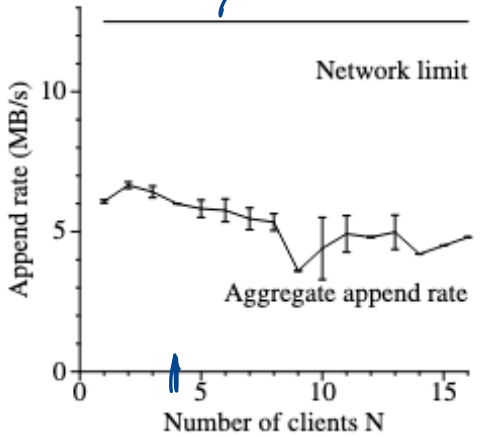
What happens with a faster network (125MB/s) but same disks (100 MB/s)?



(a) Reads



(b) Writes



(c) Record appends

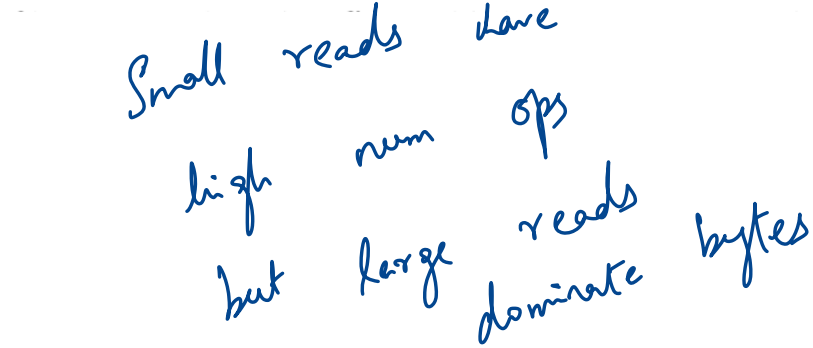
125 MB/s x 4 = 500 MB/s

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.



WHAT HAPPENED NEXT

Cluster-Level Storage @ Google

How we use *Colossus* to improve storage efficiency

Denis Serenyi
Senior Staff Software Engineer
dserenyi@google.com

Keynote at PDSW-DISCS 2017: 2nd Joint International Workshop On Parallel Data Storage & Data Intensive Scalable Computing Systems

GFS EVOLUTION

Motivation:

- GFS Master
  - One machine not large enough for large FS
  - Single bottleneck for metadata operations (data path offloaded)
  - Fault tolerant, but not HA

- Lack of predictable performance
  - No guarantees of latency
  - (GFS problems: one slow chunkserver -> slow writes)

GFS EVOLUTION

GFS master replaced by Colossus

Metadata stored in BigTable

Recursive structure ? If Metadata is ~1/10000 the size of data

100 PB data → 10 TB metadata

10TB metadata → 1GB metametadata

1GB metametadata → 100KB meta...

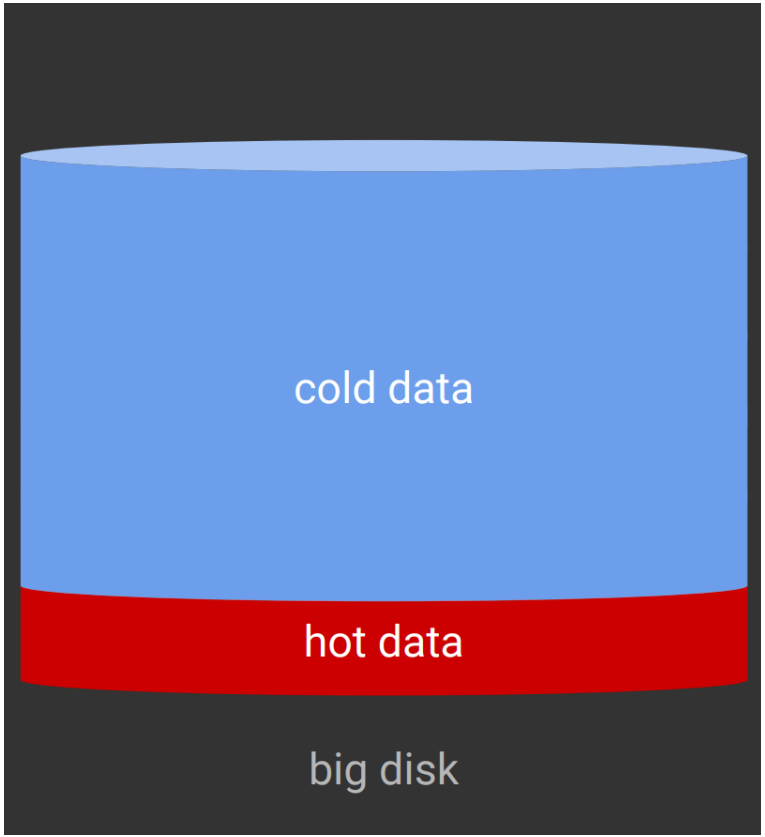
GFS EVOLUTION

Need for Efficient Storage

Rebalance old, cold data

Distributes newly written data evenly across disk

Manage both SSD and hard disks



NEW STORAGE SYSTEMS

Owl: Scale and Flexibility in Distribution of Hot Content

OSDI 2022

Later in the course!

NEXT STEPS

- Assignment 1 out tonight!
- Next up: MapReduce, Spark