Yogesh Simmhan
simmhan@iisc.ac.in

Department of Computational and Data Sciences
Indian Institute of Science, Bangalore

DS256 (3:1)

Scalable Systems for Data Science

Module 1

Introduction to Big Data & Distributed Storage

The Google File System

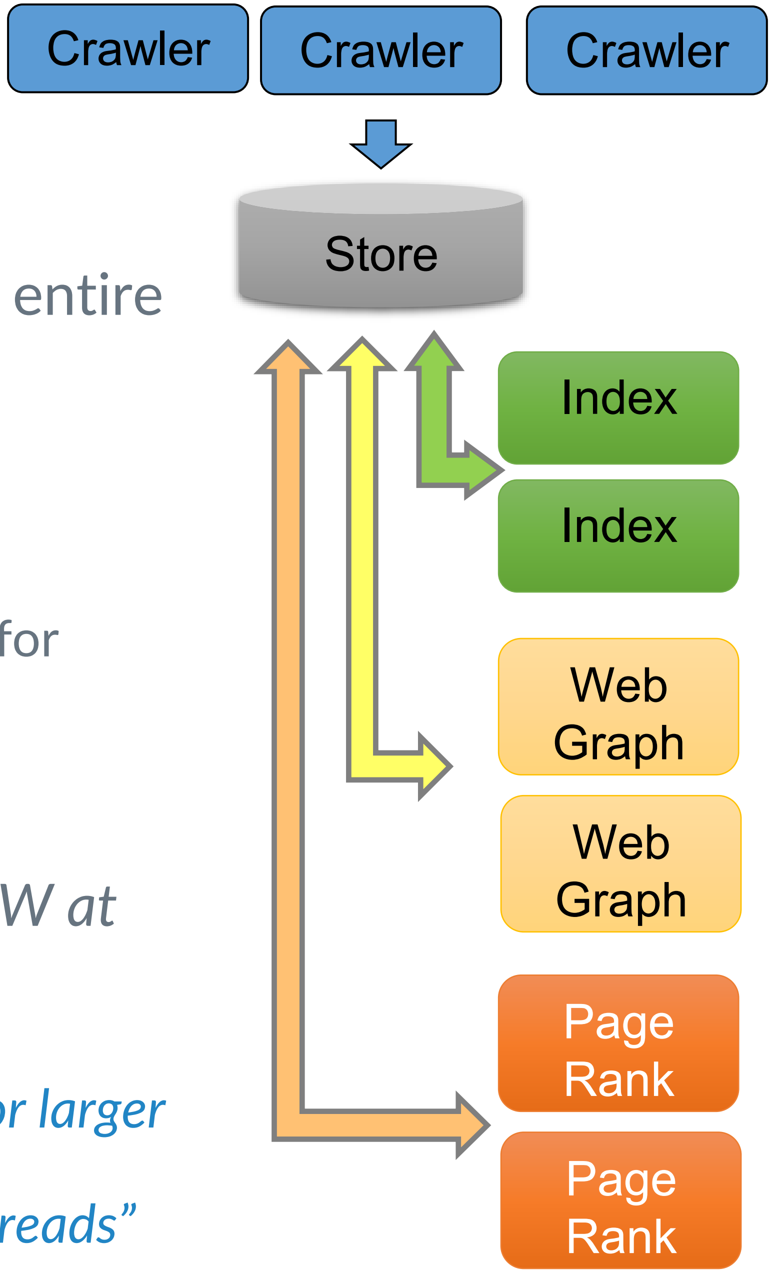
“

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

SOSP 2003

[*https://research.google.com/archive/gfs-sosp2003.pdf*](https://research.google.com/archive/gfs-sosp2003.pdf)

Motivation



- *Circa 2000*: Google wants to search the entire WWW
  - Pre-processing
    - Web crawl of millions of pages
    - Build inverted Index of keywords in Webpages
    - PageRank algorithm over web graph for ranking websites
  - Searching index & Ranking results
- *How do we store, index & search the WWW at scale?*
- How do we process this data at scale?
  - *"few million files, each typically 100 MB or larger in size"*
  - *"large streaming reads and small random reads"*

Inverted Index

URL to HTML Text mapping

u1	We the People of India, having solemnly…
u2	It was the best of times, it was the…
u3	Call me Ishmael. Some years ago…
u4	Here's my number, call me maybe…
u5	People call me the best…
u6	Number of people in India is…
u7	Best years of my life…

Parse, Tokenize

URL to Keywords Mapping

u1	We	The	People	Of	India
u2	It	Was	The	Best	Of
u3	Call	Me	Ishmael	Some	Years
u4	Here's	My	Number	Call	me
u5	People	Call	Me	The	Best
u6	Number	Of	People	In	India
u7	Best	Years	Of	My	Life

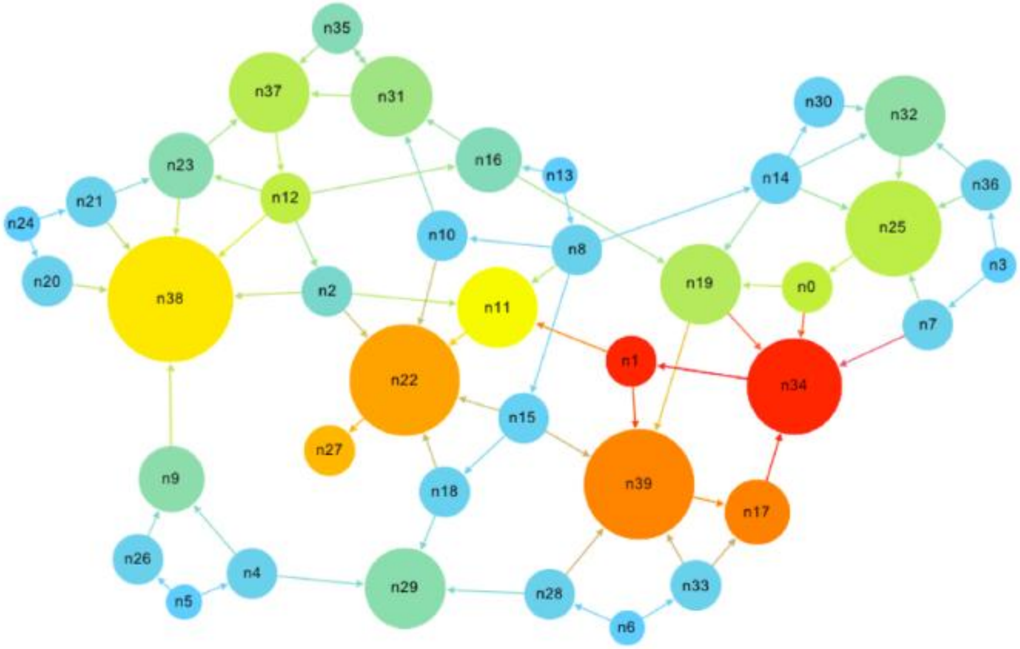
Remove stop words, contractions.
Invert index.

Keywords to URLs having keyword

People	u1	u5	u6
India	u1	u6	
Best	u2	u5	u7
Call	u3	u4	u5
Ishmael	u3		
Some	u3		
Years	u3	u7	
Here	u4		
Number	u4	u6	
Life	u7		

Web graph and PageRank

- WWW Link Graph
  - Extract links, build graph adjacency list
- Calculate PageRank

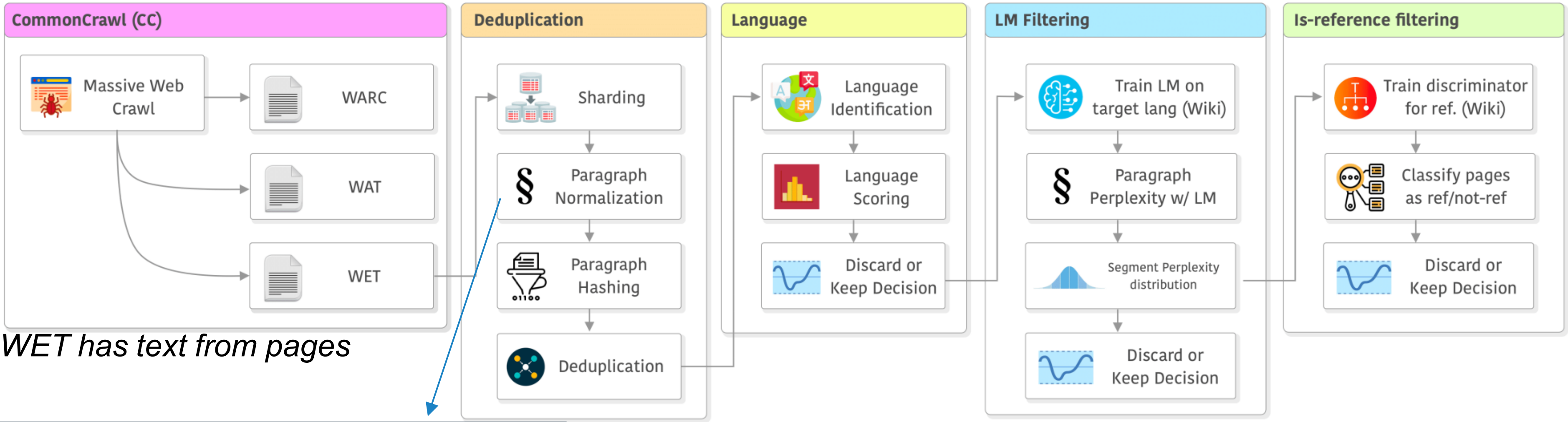


URL	PageRank
u1	0.02
u2	0.3
u3	0.08
u4	0.1
u5	0.2
u6	0.25
u7	0.05

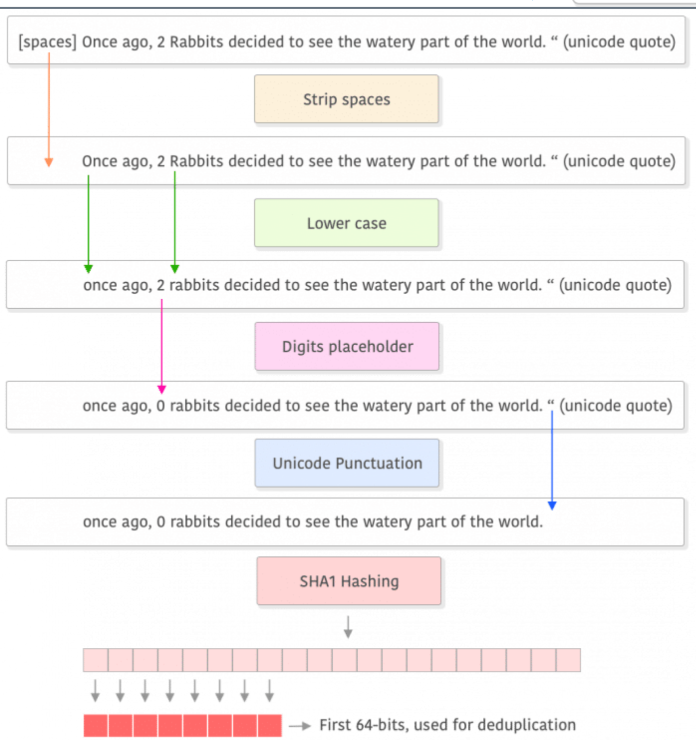
https://medium.com/@a.chachra/implementation-of-page-rank-algorithm-in-python-using-random-walk-method-d61cf136a57f

▶▶20 years

Using Common Crawl For LLaMA Training



WET has text from pages



Dataset	Sampling prop.	Epochs	Disk size
CommonCrawl	67.0%	1.10	3.3 TB
C4	15.0%	1.06	783 GB
Github	4.5%	0.64	328 GB
Wikipedia	4.5%	2.45	83 GB
Books	4.5%	2.23	85 GB
ArXiv	2.5%	1.06	92 GB
StackExchange	2.0%	1.03	78 GB

https://blog.christianperone.com/2023/06/appreciating-llms-data-pipelines/
LLaMA: Open and Efficient Foundation Language Models
The RefinedWeb Dataset for Falcon LLM

Motivation for GFS

- DFS with reliability, performance and scalability
- Which can store and access large datasets
- That are written once and read often
- For (mostly) sequential reads and some random reads
- On commodity servers

Motivation for GFS

- Inexpensive commodity components that **often fail**
  - *Detect, tolerate, and recover promptly* from failures on a *routine basis*.
- Modest number of **large files**
  - A few million files, each >100 MB.
  - Multi-GB files common case. Small files need not be optimized
- **Read workload**
  - **Large streaming reads**: Each op reads >1 M; Same client will read contiguous region of a file.
  - **Small random read**: Reads a few KBs at some arbitrary offset. Apps often batch and sort their small reads to do sequential access

Motivation for GFS

- **Write Workload**
  - Many large, sequential *append-only Writes*, like reads
  - Files are *seldom modified*
  - Small random writes supported but *need not be efficient*

- Well-defined semantics for **multiple concurrent clients**
  - Append to the same file by hundreds of (producer) clients
  - Atomicity with minimal synchronization overhead
  - (Consumer) client may overlap reads with writers

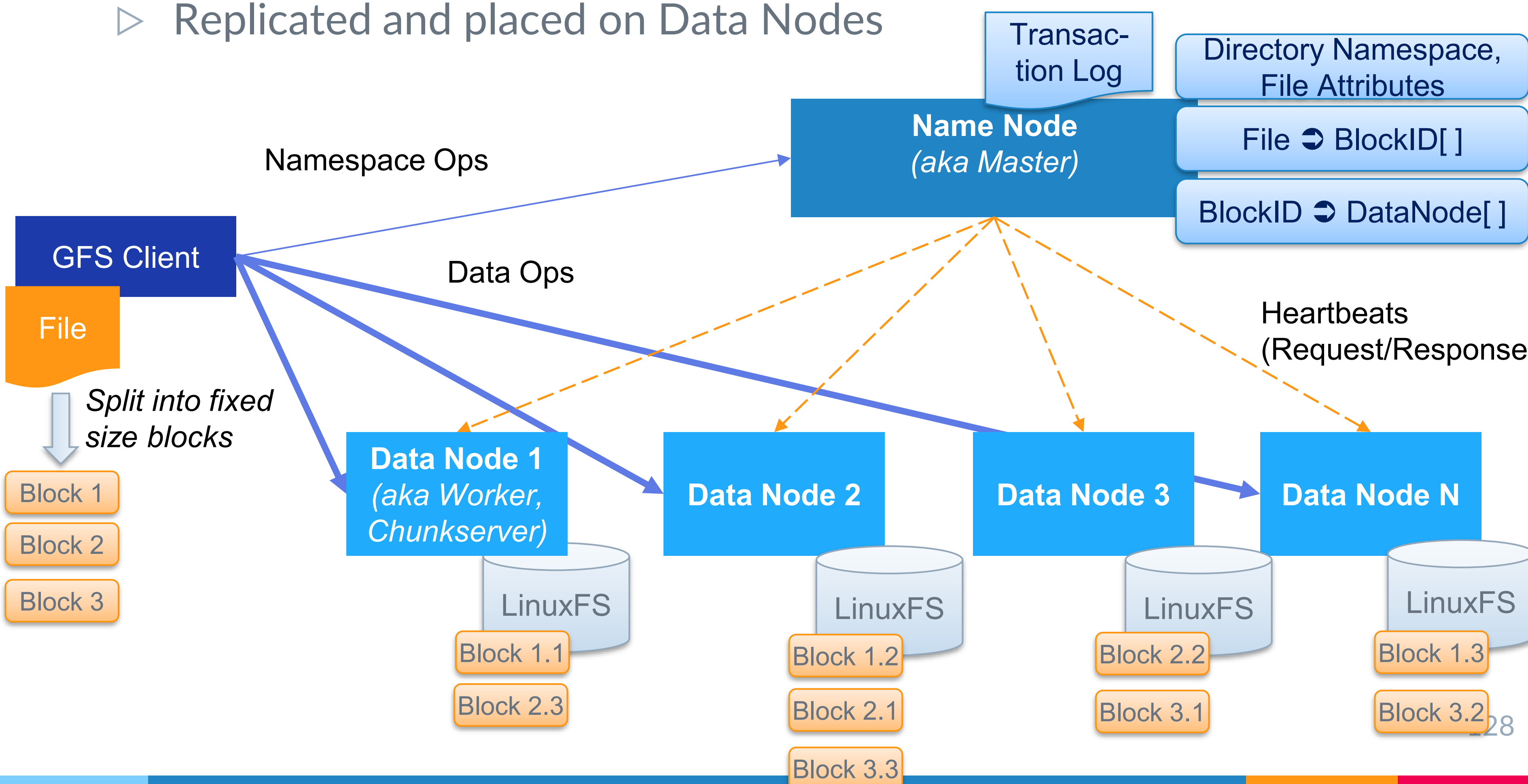
- **High sustained bandwidth** is more important than low latency
  - Batch vs. Interactive

POSIX API

- Files organized *hierarchically* as directories
- Client API: *create*, *delete*, *open*, *close*, *read*, *write files*
- New Operations
  - Snapshot
  - Record Append to support concurrent and atomic writes

Architecture

- One *Master* (Name Node in HDFS)
- Many *Chunk Servers* (Data Nodes or Workers in HDFS)
- *Clients* can be anywhere, including on Chunk Server
- Files partitioned into fixed size blocks (or Chunks)
- Replicated and placed on Data Nodes



Architecture

- One *Master* (Name Node in HDFS)
- Many *Chunk Servers* (Data Nodes in HDFS)
- *Clients* can be anywhere, incl. on Chunk Server

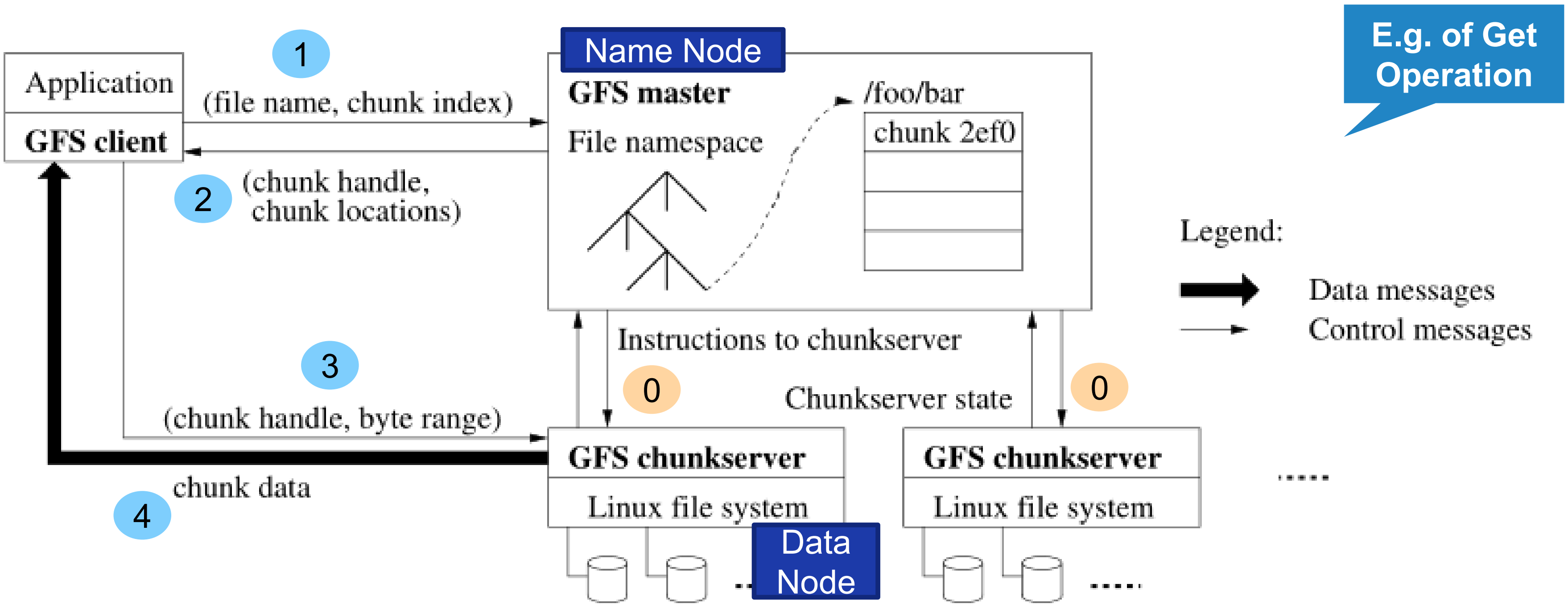


Figure 1: GFS Architecture

Architecture

- *Files* divided into *chunks* (*blocks*) with unique id
  - Stored on Data Node
  - Stored in *Linux file system* as a regular file
  - Read/write of *(chunk-file, offset)*
- Each block has n=3 *replicas* by default

Why?

- Name Node maintains *namespace metadata*
  - *Directory struct, access control*
  - Mapping *files to block IDs; block IDs to Workers*
- Name Node sends *Heartbeats* to Data Nodes
  - Piggyback: *Sent instructions. Receive states* as response.
- Client interacts with Name Node for metadata ops

Why?

- Client interacts with Data Node for data ops

Why?

- No *data* caching is done

Why?

Block Size

- 64MB (128MB in HDFS v3)
  - Much larger than filesystem block of ~4kB
  - Plain Linux file on filesystem
  - Only bytes used by blocks are stored on disk, extended on demand
    - Need to keep allocating more file blocks, but avoids over-allocation on disk

Discuss

- Pros & Cons of Large block sizes
  - Reduce metadata size on Name Node, **O(# blocks)**
  - Avoids frequent client communications with Name Node
    - Allows large metadata caches at client
  - Single persistent TCP connection to Data Node
    - Smaller overheads

Master Metadata

- In-memory data structures
  - File namespace
  - *File to Block* mapping
  - *Block to Data Node* mapping

- In-memory helps with fast/complex ops
  - Garbage collection, re-replication, load-balancing
  - Compact memory use
    - 64 bytes for each block and per file

Metadata: Block Mapping

- Block to Data Node mapping built on-demand at startup
  - Can be *reconstructed* at any time
  - *Fault tolerance*, less consistency issues with Data Node flux
  - Data Node is the **final arbiter** of blocks it has
    - Data may “spontaneously vanish”, e.g., data loss due to disk failure, etc.

- Name Node **maintains** mapping after initial startup
  - It controls block placements, etc.

- Block placement, monitoring using Heartbeats

Metadata Operations Log

- *Only persistent record of metadata!*
- Historic record of critical metadata changes
  - Namespace ops, File-to-Block mapping
  - Needs resilience on Name Node failure
  - **Logical timeline**

- Ops log is **persisted** (on all replica name nodes) before metadata update visible to client
  - Loss of metadata → Loss of entire file system

- Replicated on multiple name nodes
  - **Periodic checkpoint** of metadata state to disk
    - **Fast** checkpointing using new thread/log file without stopping active operations
    - Compact **B-Tree** that maps to memory without parsing
  - Recovery of metadata by **replaying ops log** since last checkpoint

Name Node's responsibilities

- Garbage Collection
  - Simpler, more reliable than traditional file delete
    - Name Node logs the deletion in namespace
    - *Renames* the file to a hidden name
    - *Lazily* garbage collects hidden files

- Stale replica deletion from Data Node
  - Detect "stale" replicas using block version numbers

Fault Tolerance

- High availability
  - Fast recovery
    - Name and Data Nodes are *restartable* in a few seconds
  - Block replication
    - Default: 3 replicas.
    - Replication for fault tolerance vs. Replication for performance
  - Shadow Name Node

- Data integrity
  - Checksum for every 64kB block in each chunk