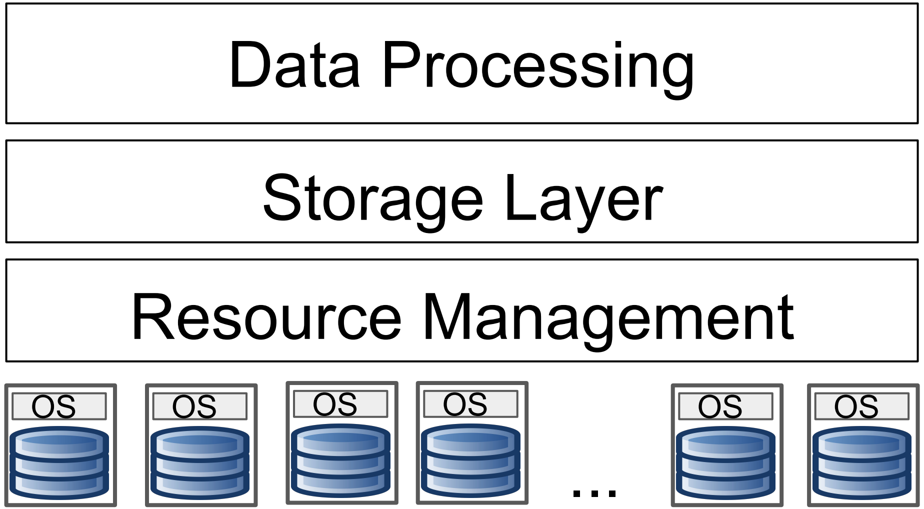
Databases and Big Data

Hadoop 1

Recap

- Cloud computing runs your Big Data
- Reliability comes from software
- Many challenges:
  - Failures
  - Consistency
  - Scalability
  - Utilization



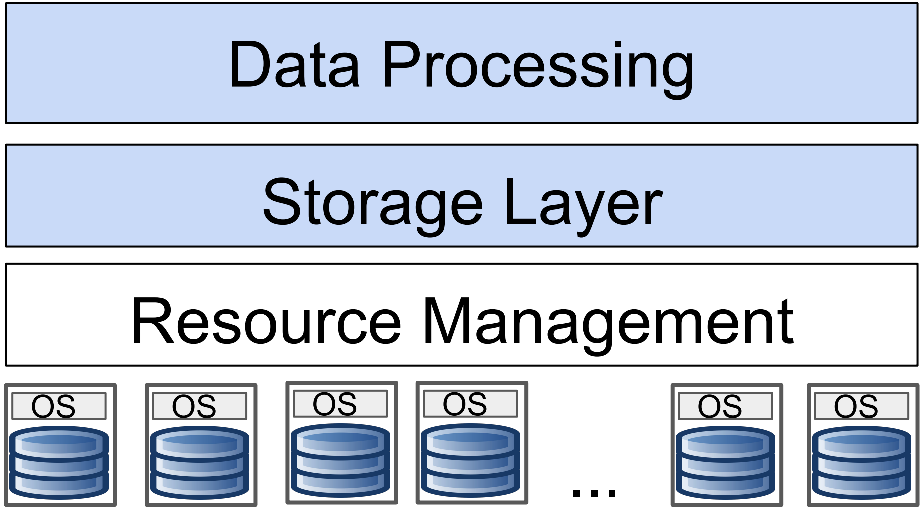
Data Processing

Storage Layer

Resource Management

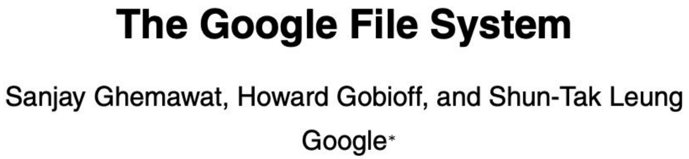
This week

- Hadoop
  - Storage and Data Processing layer
  - How it scales
  - How it tolerates failures



History

- 2003: Google published GFS

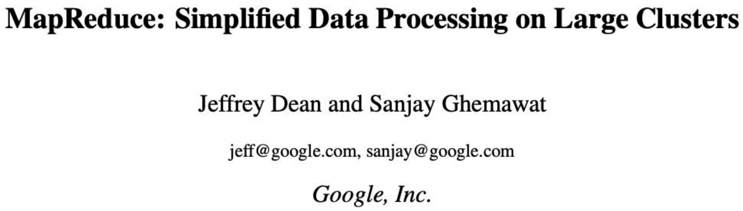


The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

- 2004: Google published MapReduce



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

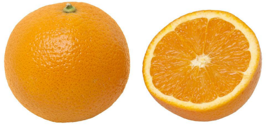
- 2006: Hadoop started
    - As open-source implementation of GFS, MapReduce

Why Distribute File System (HDFS)?

Why?

- What was the problem?

There’s Big Data



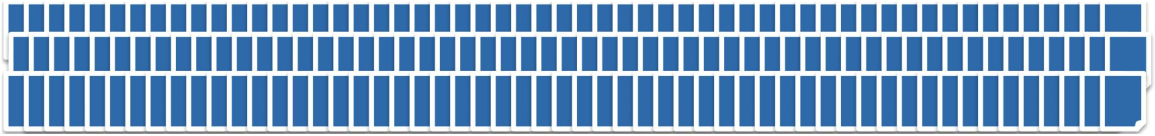
And

There’s Big Data

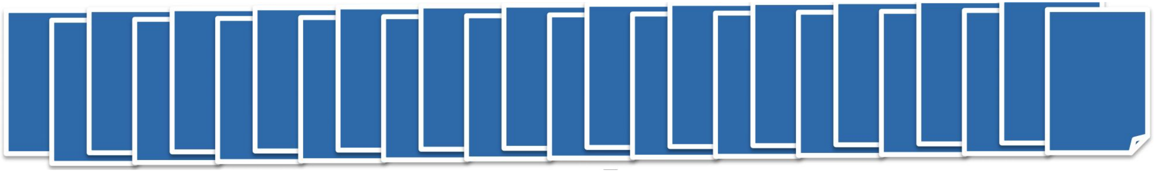
Why?

- Big Data

Huge number of files

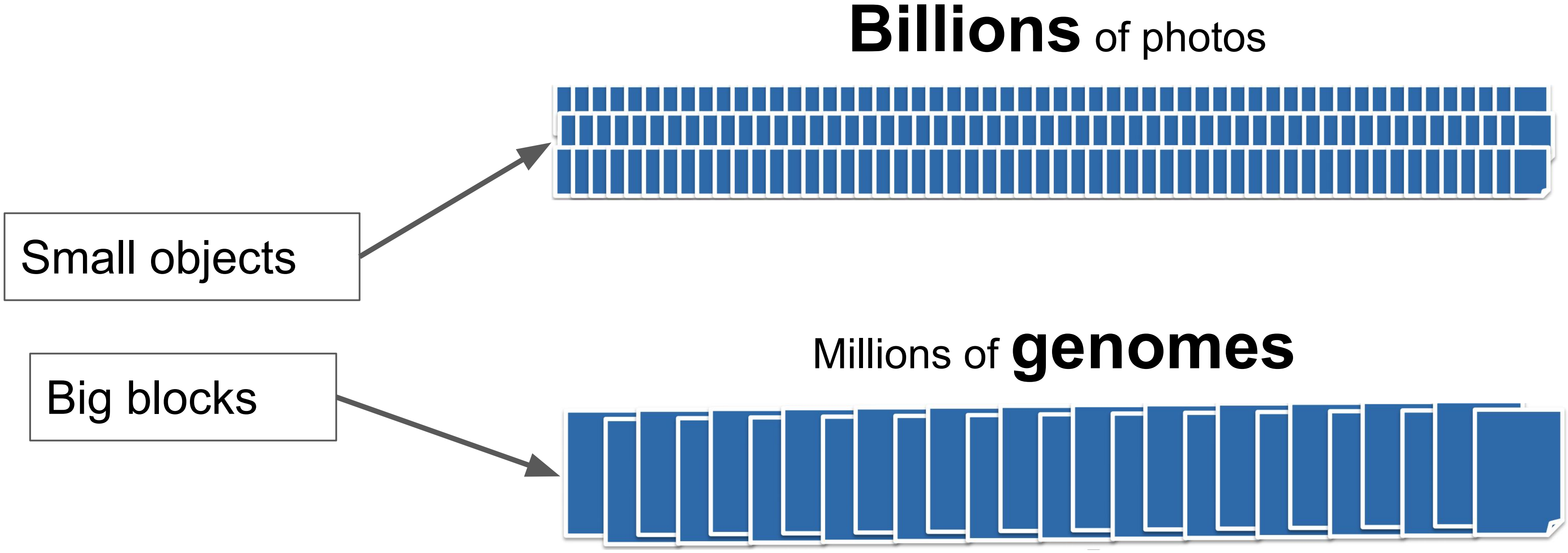


A number of huge files



Why?

- Big Data



Billions of photos

Small objects

Millions of **genomes**

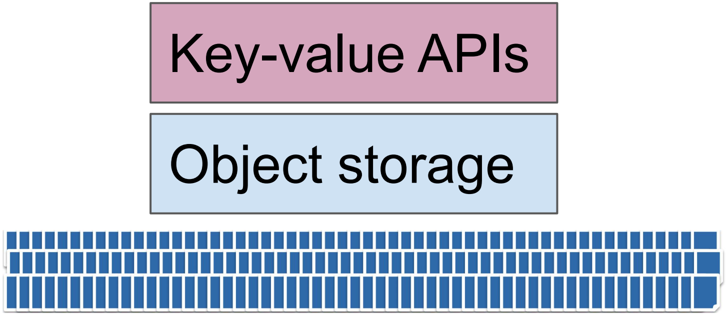
Big blocks

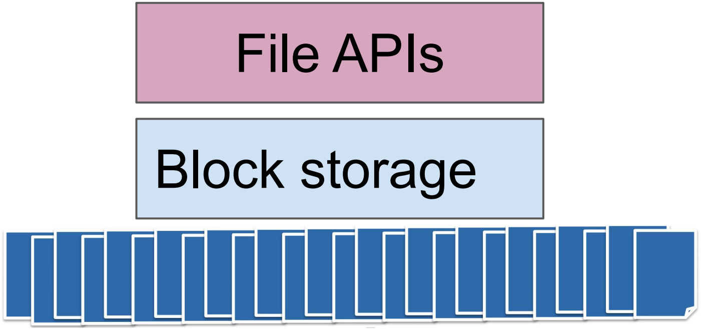
Why?

- More generally:

like redis, memcache to handle billions of small data

key-value storage is not sufficient if you wanna store big files





Raw data

Derived / aggregated data

Why?

- Challenges Google faced (then, in 2003)
  - Databases were expensive
  - Non-table data
  - 10s of PB
  - 100-200MB/s disk throughput: 2 hours to read 1TB
- But disks were cheap: $50/PB

Google mainly has non-relational data

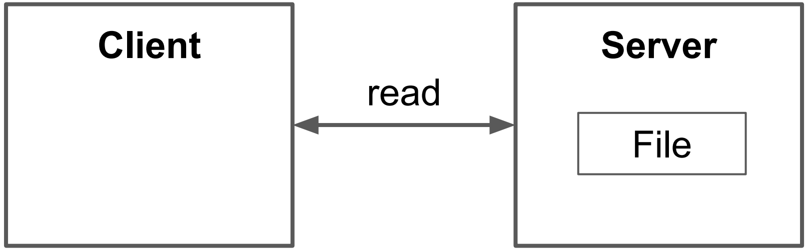
Why?

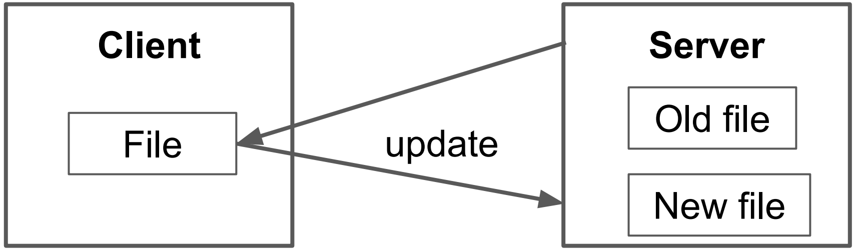
- What were available then?
  - ~~RDBMS~~
  - Distributed file systems?

*Server is main **bottleneck***
All operations and traffic goes to one single server.
Server will fail if there is alot of traffic going in

Network File System (NFS)

Problems?

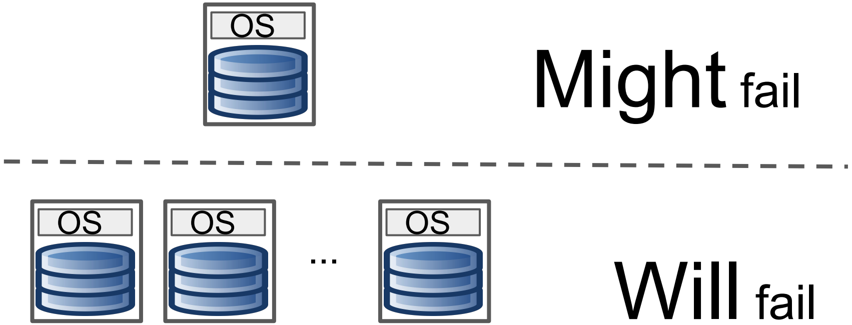




Why?

- New requirements for a file systems
  - 1000s of clients read/write to/from 1000s of disks
  - Failure
  - Size:
    - Linux files: < GBs
    - Google files: **TBs**

Cannot afford 4K-blocks like Linux files!



Might fail

Will fail

Why?

- In summary

NO Systems

can do

all these

Support many clients

Support many disks

Support many PBs

Robust under failures

Read/write like files

Why?

- Assumptions

Support many clients

Support many disks

Support many PBs

Robust under failures

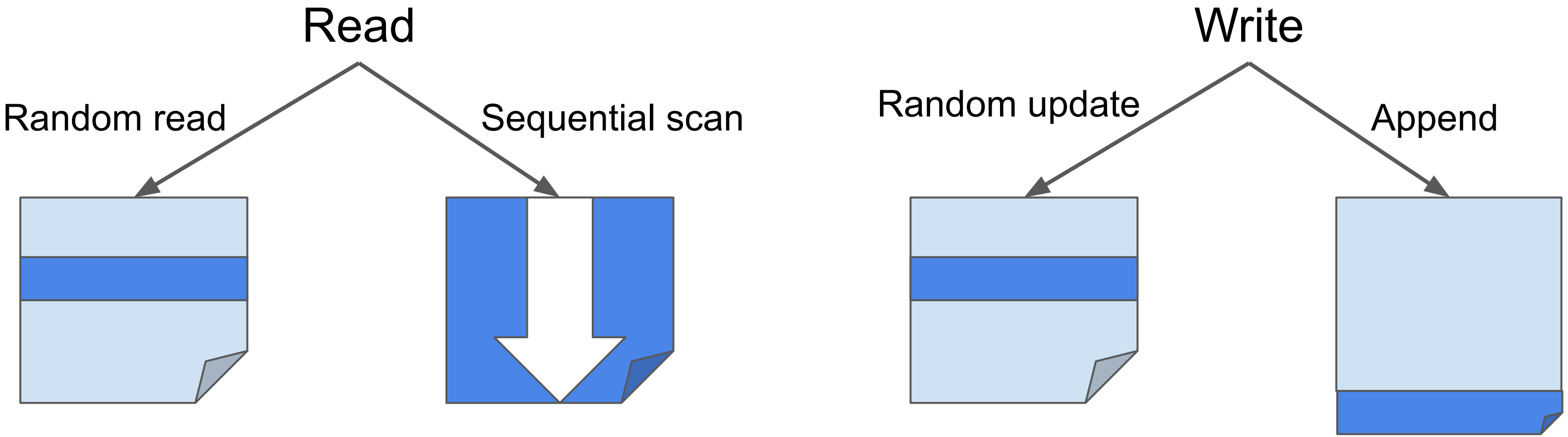
~~Read/write like files~~

e.g. google internal system can remove support many clients

Why?

- If not like a normal file, then what?
- Google’s files:
    - **Read sequentially**
    - **Append only**

no more random read/update

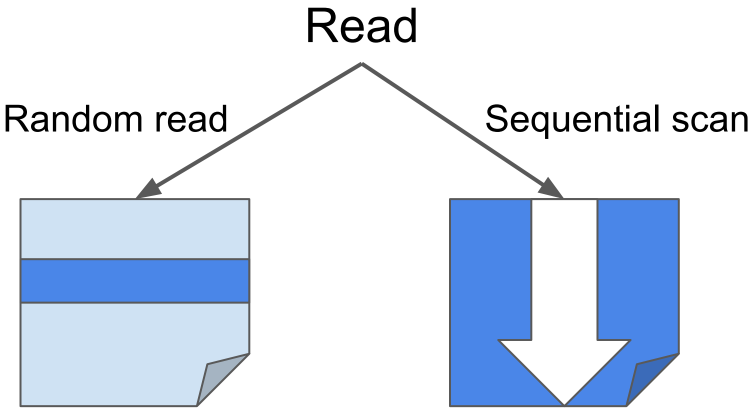


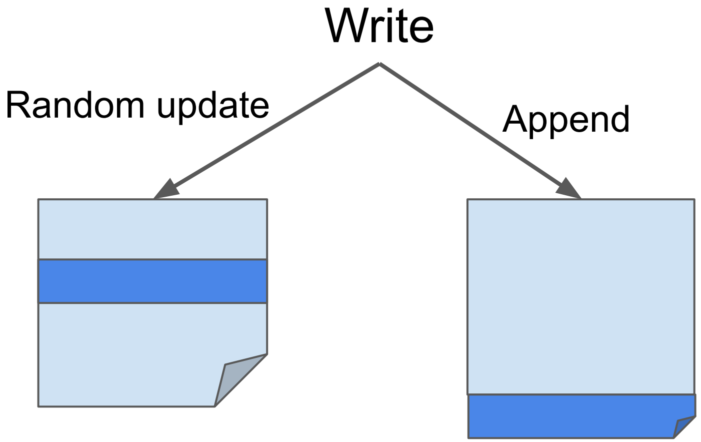
Why?

- Files are:
  - **Read sequentially**
  - **Append only**

Some applications have this pattern:

- + Event logs
- + Web crawling
- + Sensor data
- + etc.





Summary

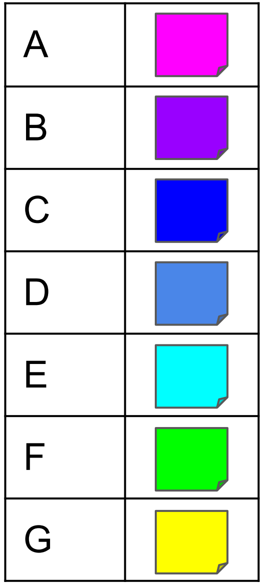
- Google File System (GFS) hugely influential
- Hadoop Distributed File System (HDFS) implements GFS
- GFS changed all existing DFS assumptions on its head!
    - **A new era of data center computing!**

HDFS Internal

File System

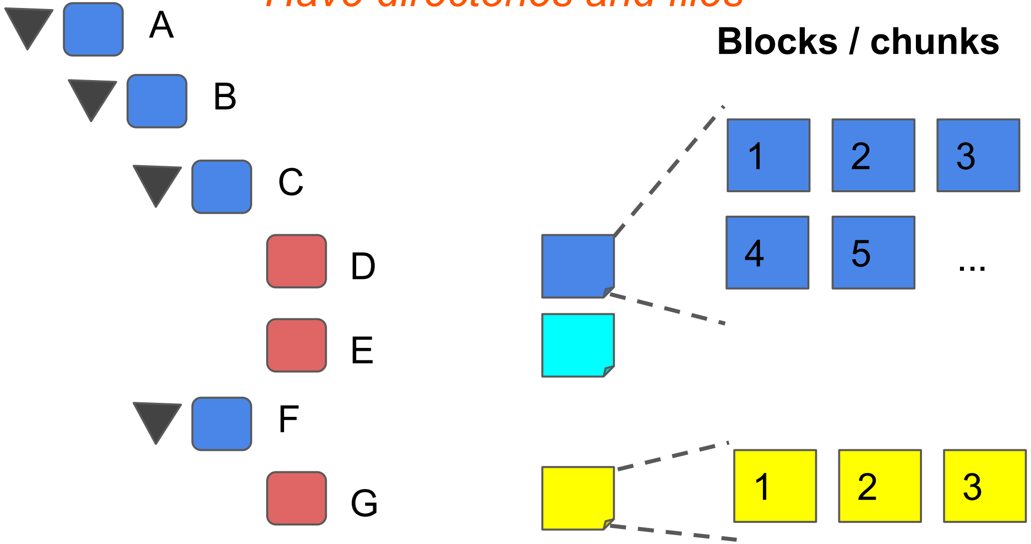
- File system model

No structure



Flat namespace (key-value store)

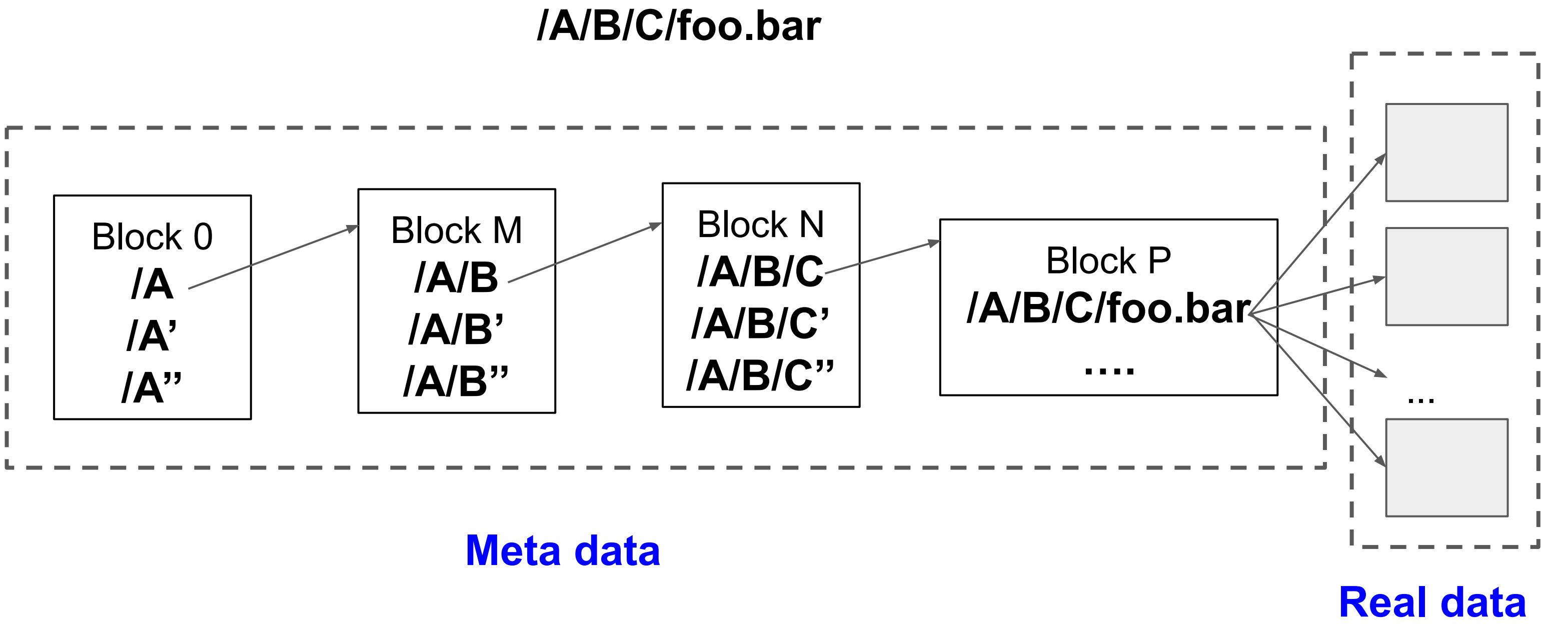
Have directories and files



Hierarchical namespace (file system)

File System

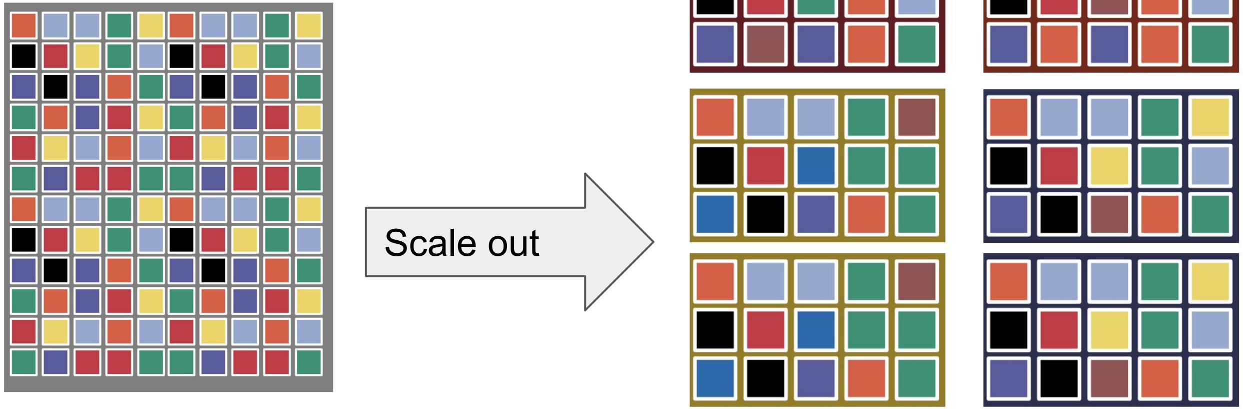
- How a file system works (roughly)



HDFS

need to also keep track of which block is at which machine
on top of which block makes up which file

- Why blocks?
  - Simple abstraction
  - A file may be bigger than a disk

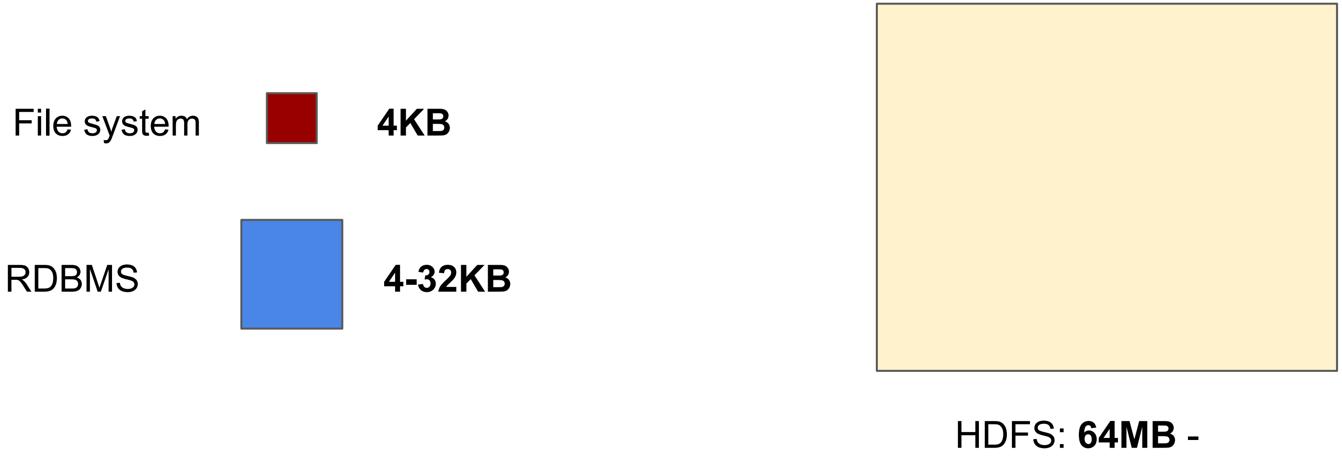


Single machine

Distributed file systems

HDFS

- Block size



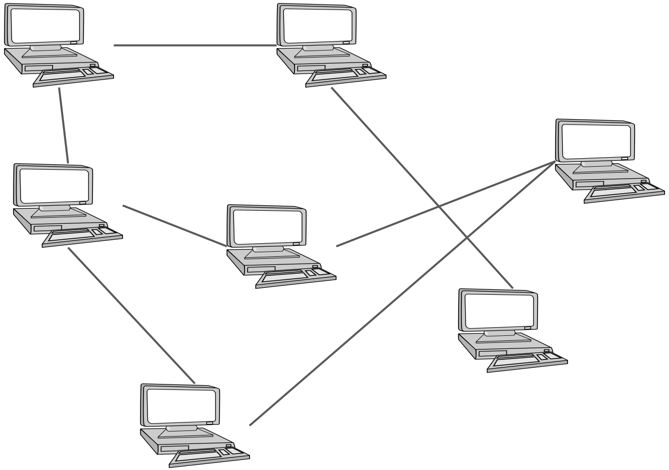
File system 4KB

RDBMS 4-32KB

HDFS: **64MB -**

HDFS

- Architecture
  - Peer-to-peer?

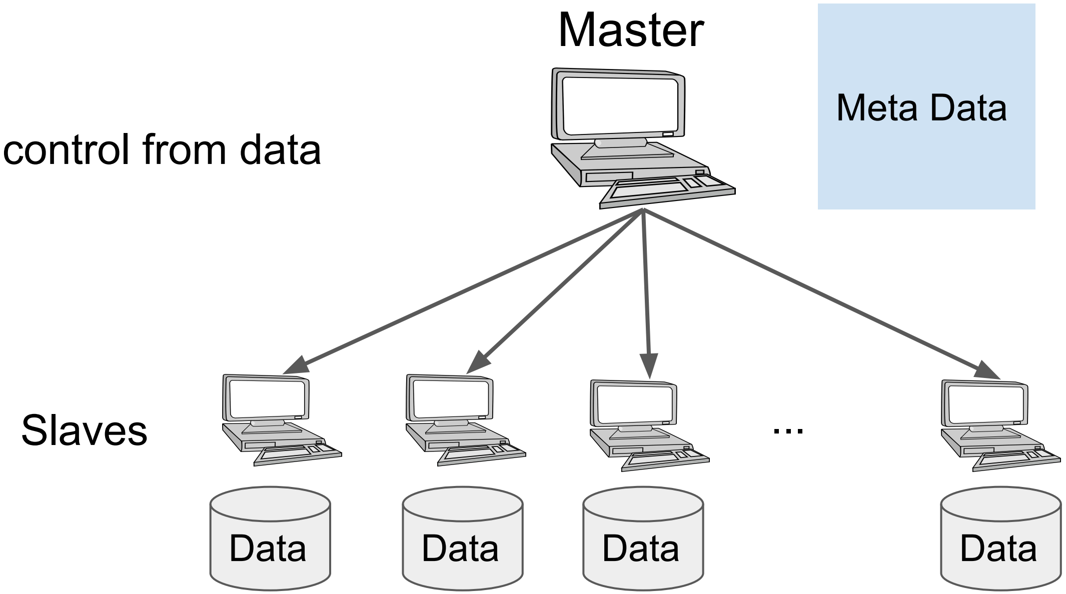


Suitable for something like file sharing (Bittorrent)

But if you own all these machines, you can do (**much**) better

HDFS

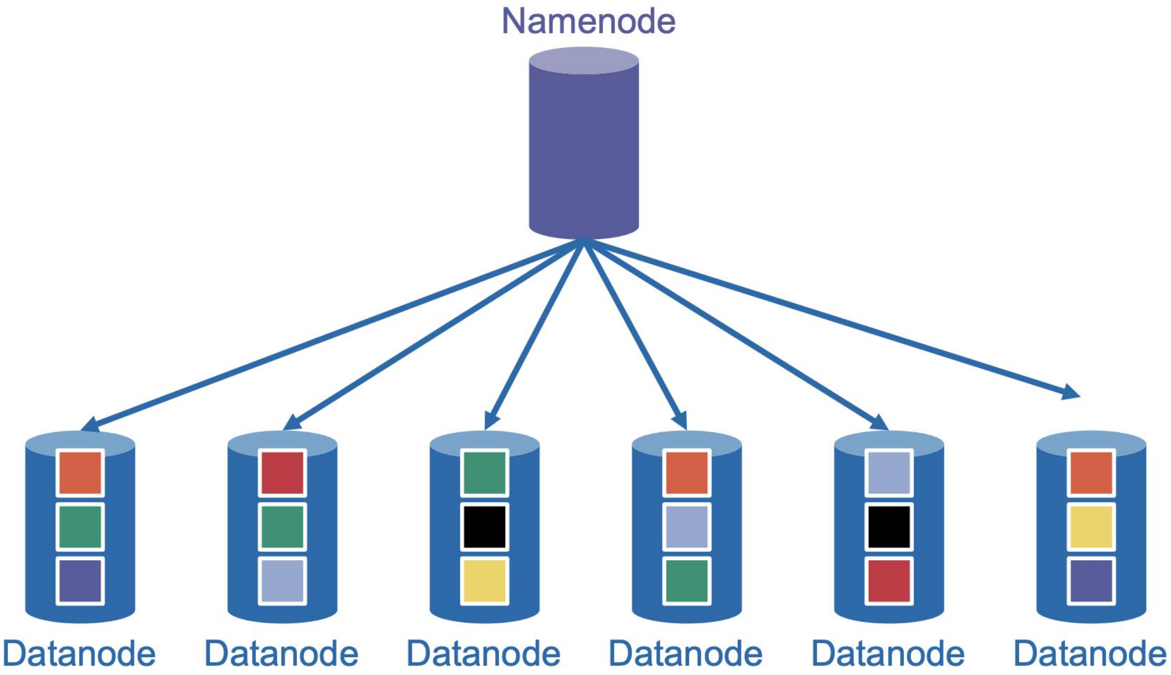
- Achirecture:
    - Master-slave
        - Separate control from data



HDFS

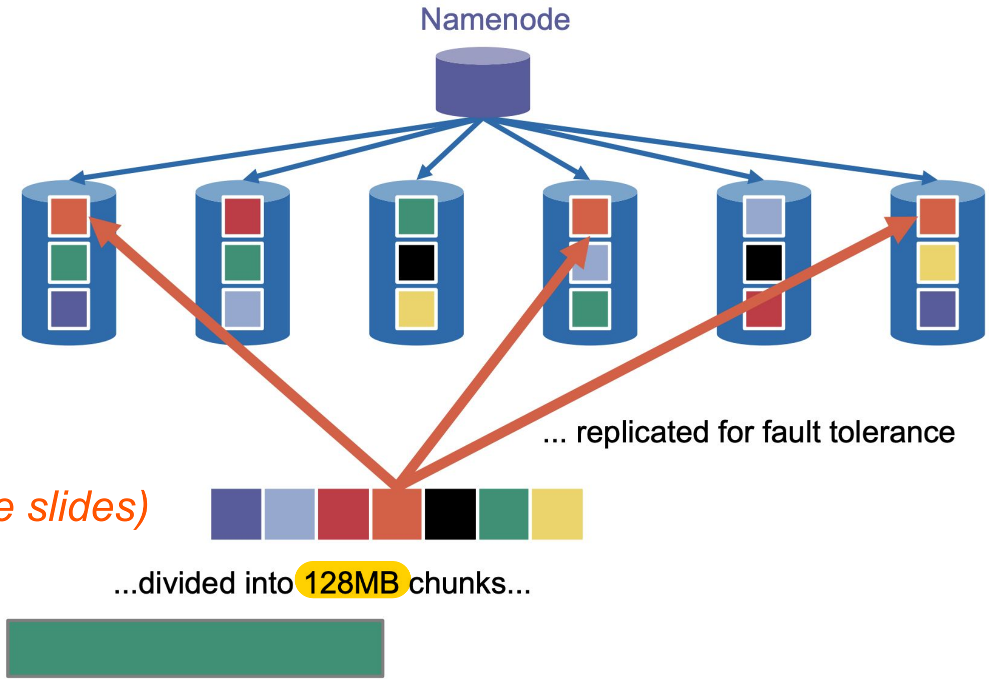
- Architecture

contain the namespace (metadata) about the data nodes



HDFS

- Architecture:
    - File’s perspective



Namenode

... replicated for fault tolerance

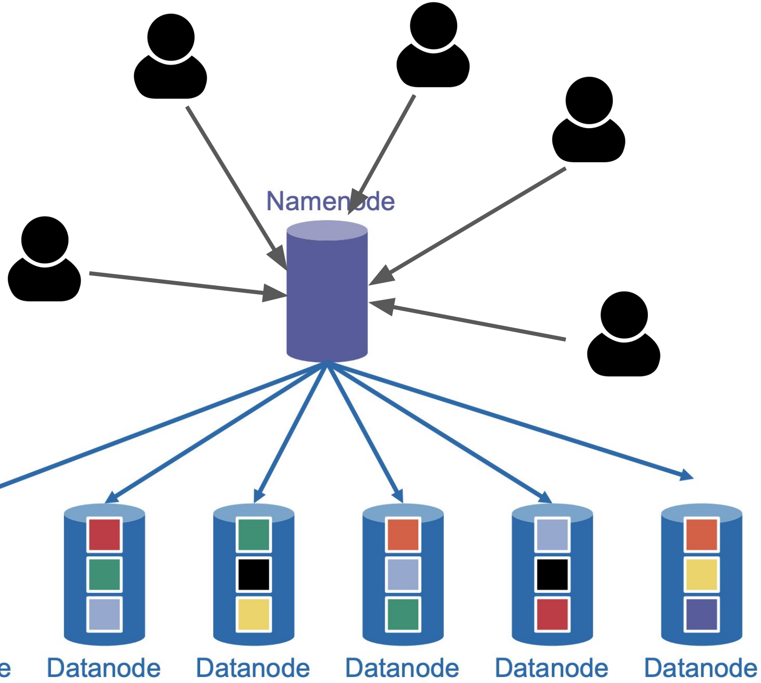
...divided into 128MB chunks...

File...

used to be 64mb blocks (from before slides)
***replication** to deal with failures*

HDFS

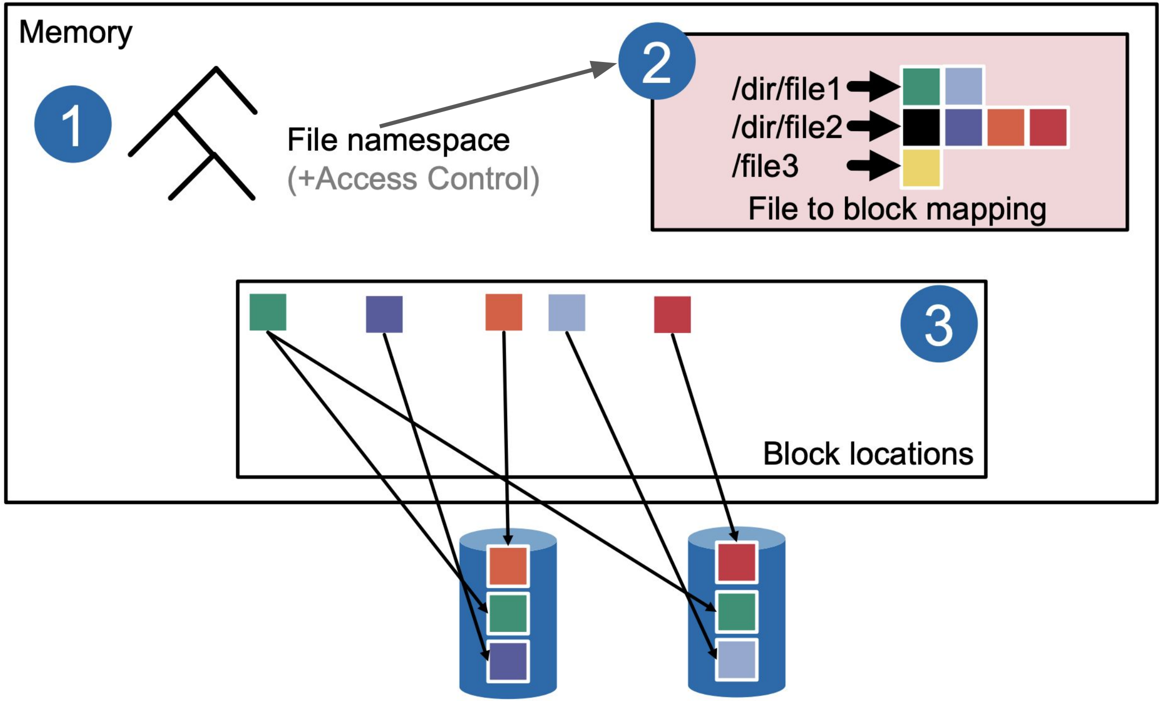
- Architecture
  - User’s perspective



HDFS

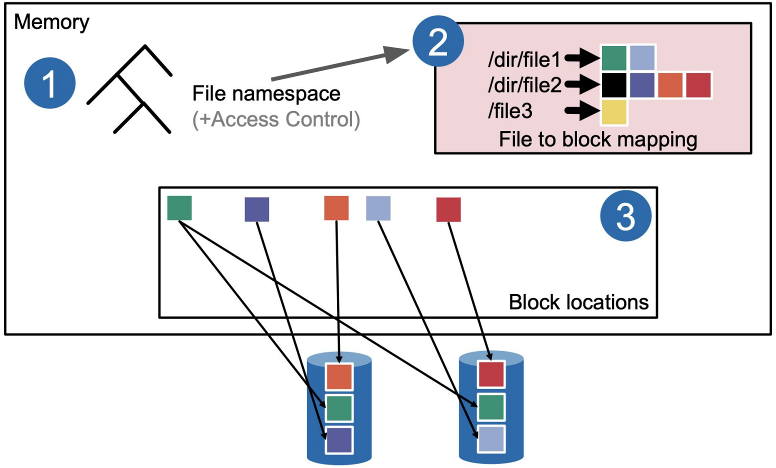
- Inside the NameNode:

Meta data



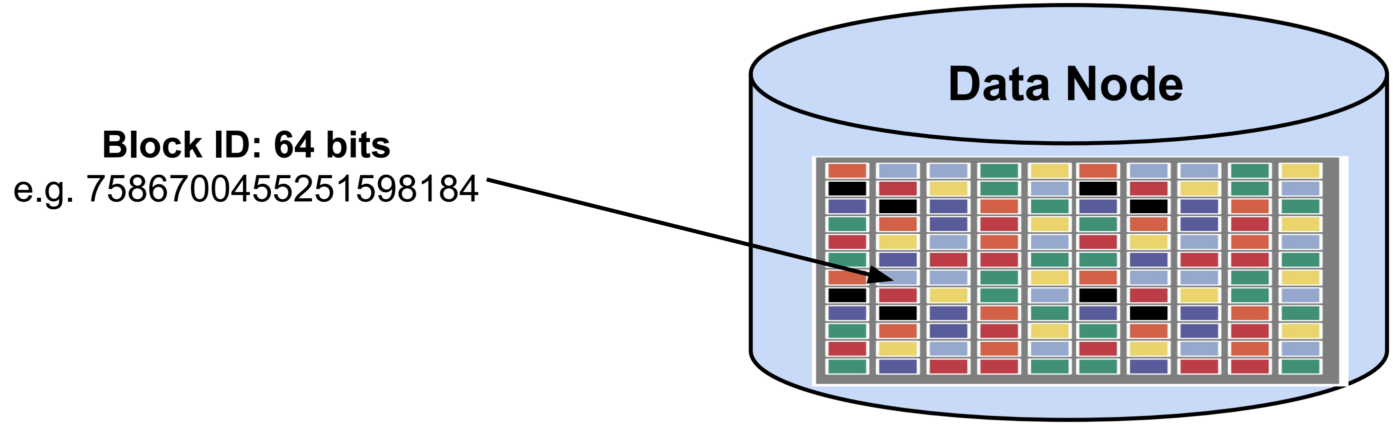
HDFS

- How to lookup a file: /foo/bar
    - Use (1) to resolves /foo/bar to (2)
    - Use (2) to get block IDs
    - Use (3) to get block locations



HDFS

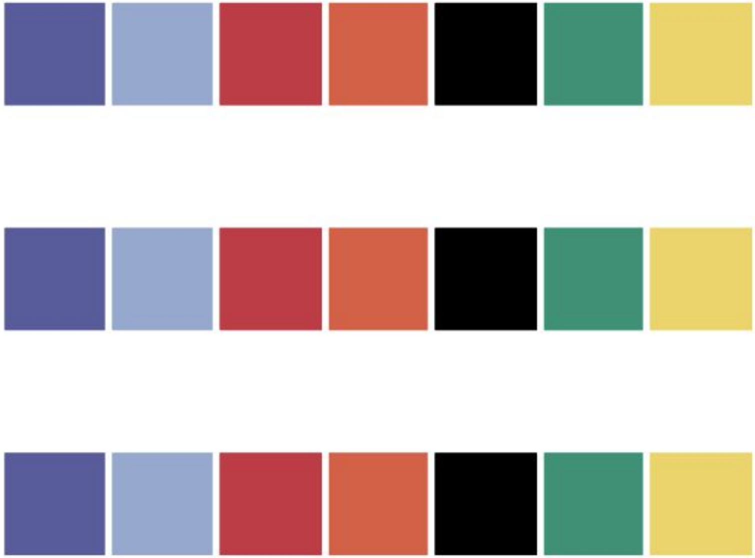
- Inside a data node
    - Blocks stored on local disks
    - Each has unique, global IDs



HDFS

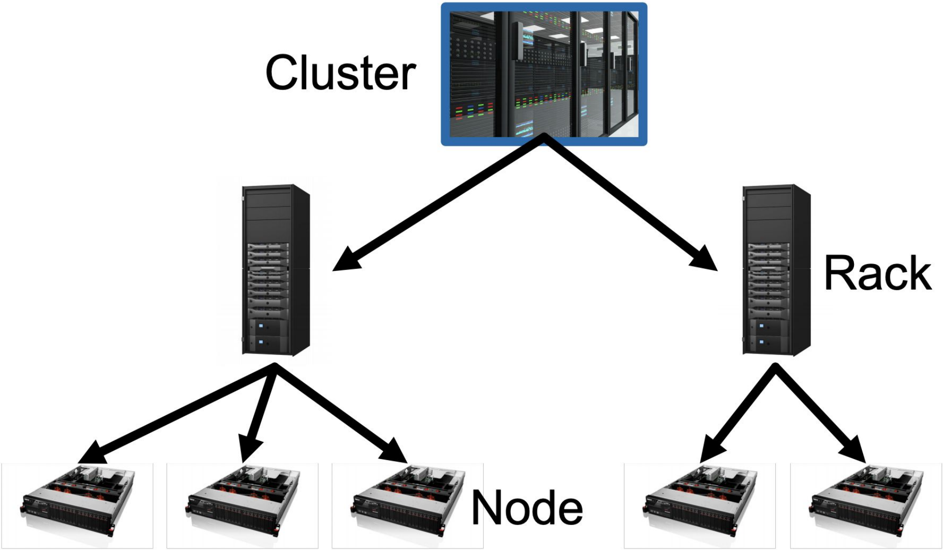
- Replicas:
  - To survive failures
  - At block granularity

Default # replicas: **3**



HDFS

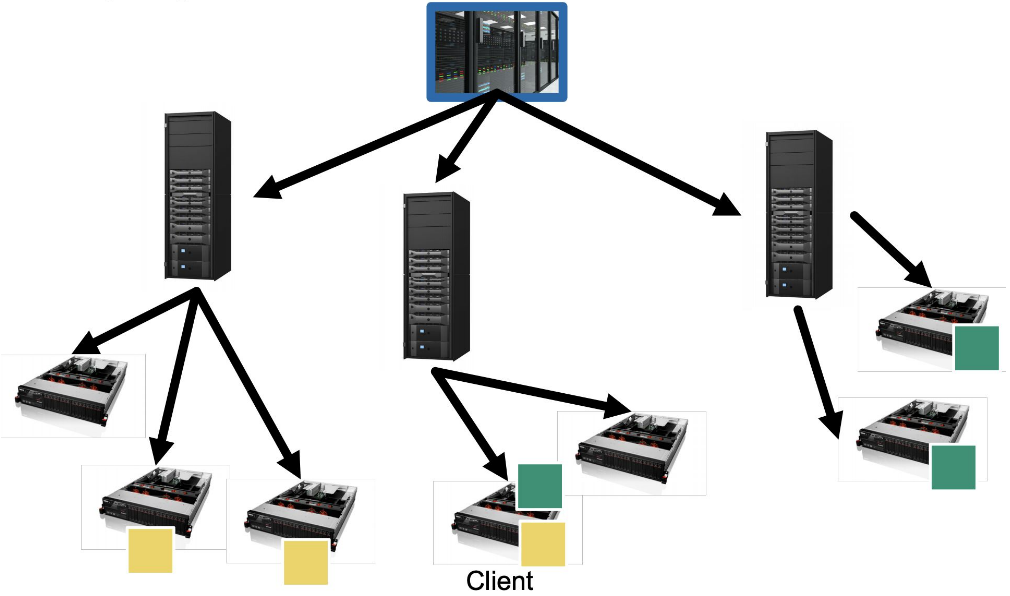
- Replicas question: location location location
  - Which nodes?
  - Which racks?
- Goals:
  - Maximize chance of survival
  - Also maximize load balance



HDFS

- Replica placement strategy:
  - Replica 1: rack A (first DataNode contacted during write)
  - Replica 2: different rack B
  - Replica 3: rack B
  - Replica >4: random

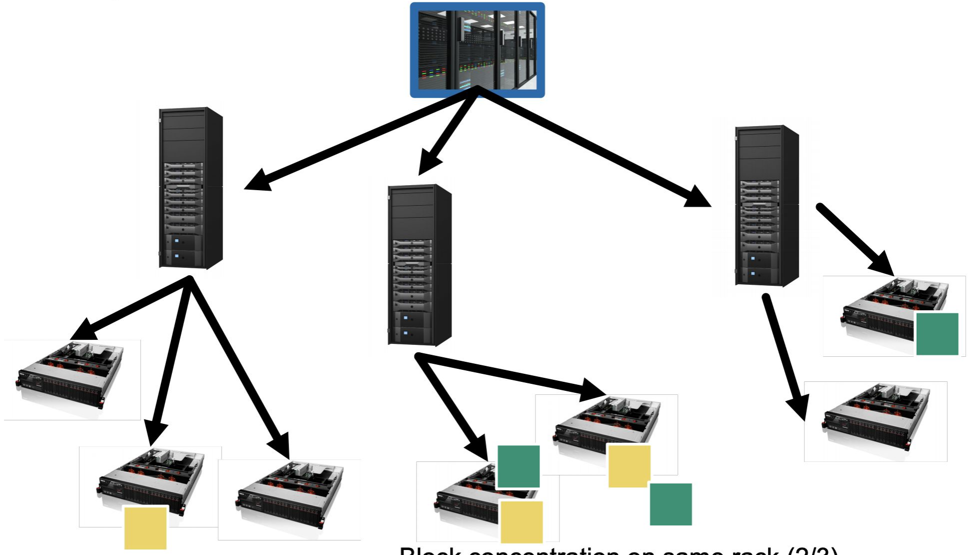
1 -> A
2 -> different rack
3 -> same rack as in (2)
(Shown to work well)



HDFS

- Replica placement strategy:
    - Replica 1: rack A (first DataNode contacted during write)
    - Replica 2: different rack B
    - Replica 3: rack B
    - Replica >4: random

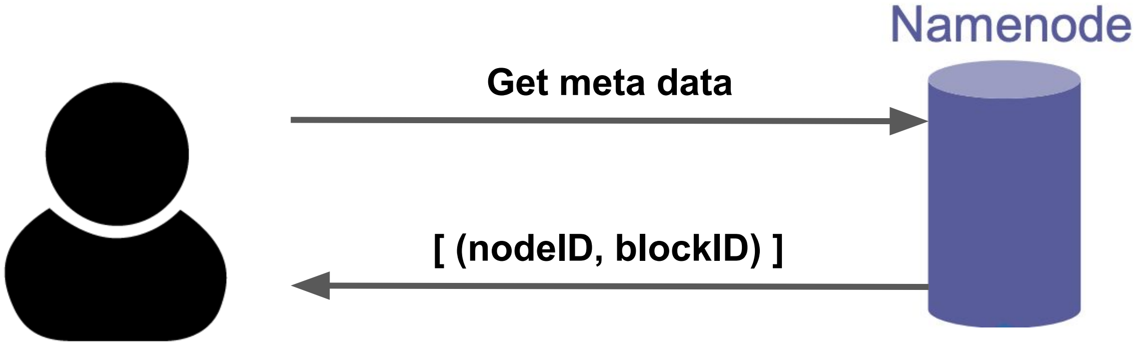
What happen when Replica 1 and 2 in the same rack?



Block concentration on same rack (2/3)

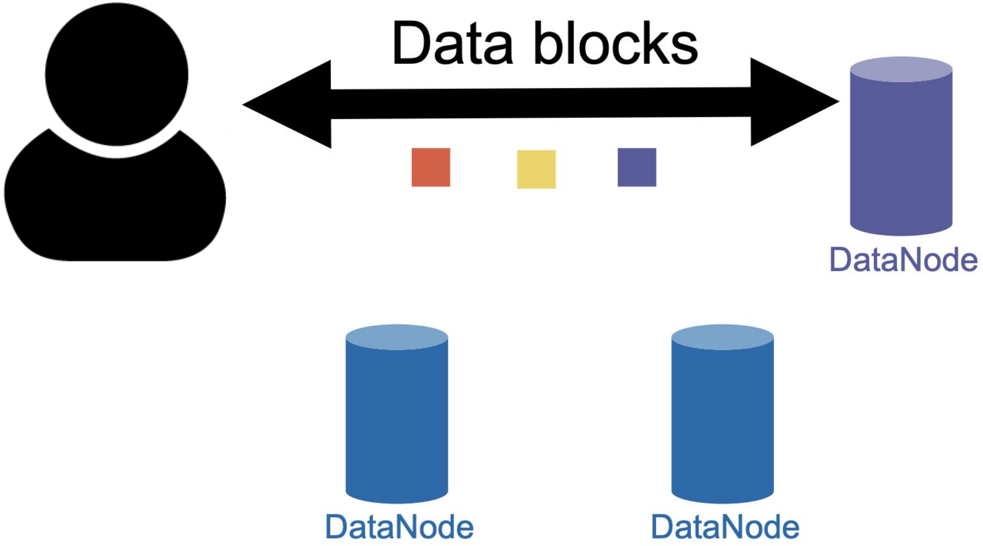
HDFS

- Client protocol:
  - Step 1: open the file
    - To read
    - Or write



HDFS

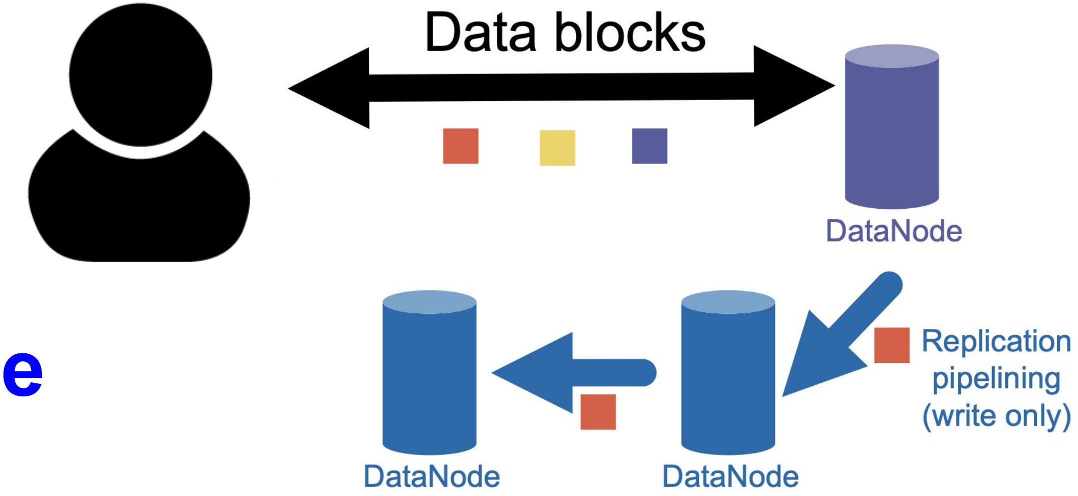
- Client protocol:
  - Step 2: direct transfer to/from DataNodes



Read

HDFS

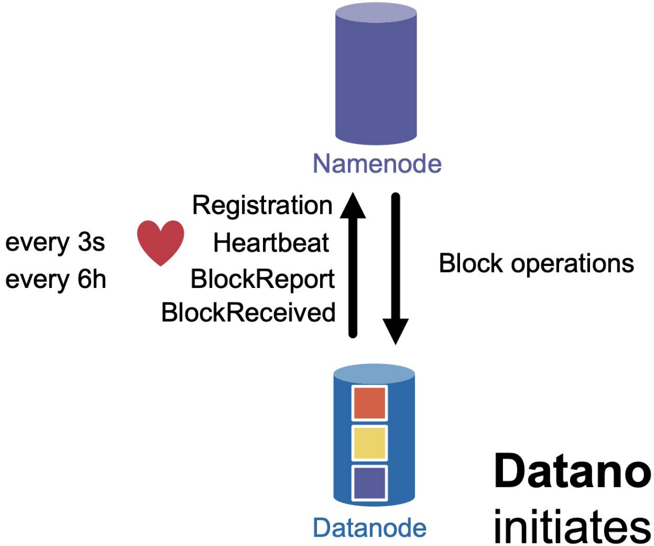
- Client protocol:
  - Step 2: direct transfer to/from DataNodes



Write

HDFS

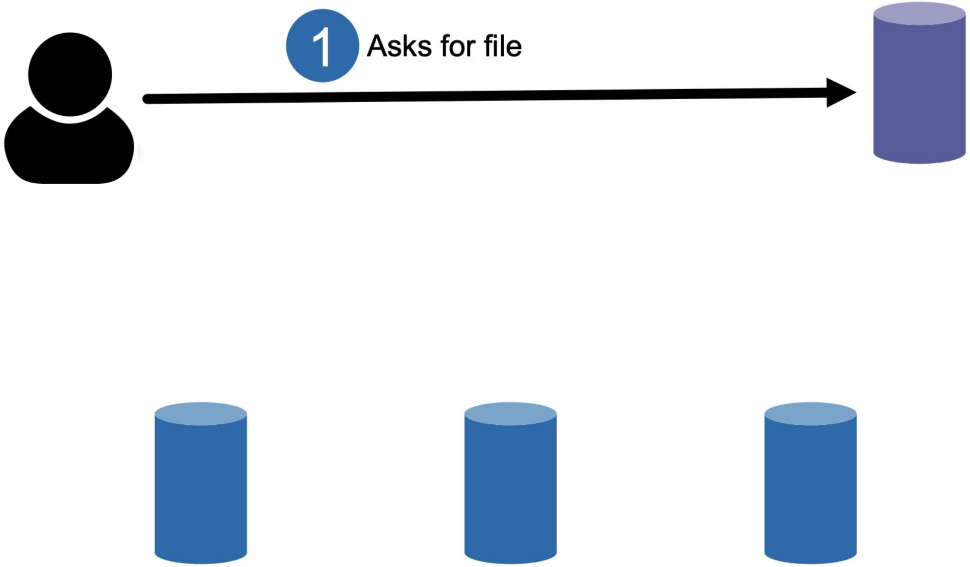
- Client protocol
  - Namenode must monitor blocks
    - Else, how does it know if a block is still available (DataNode still alive)?



Datanode always initiates connection!

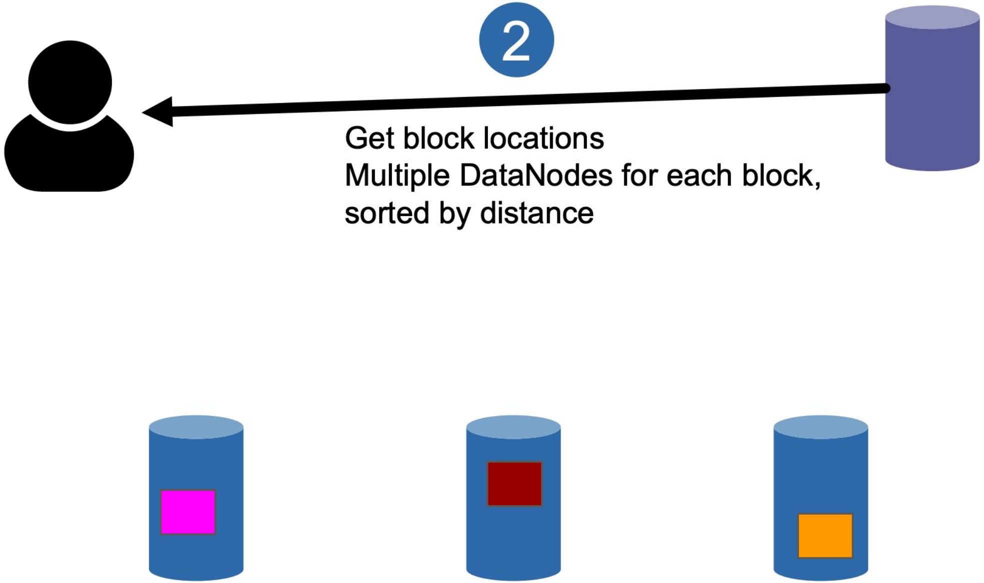
HDFS

- Detailed protocol: Read



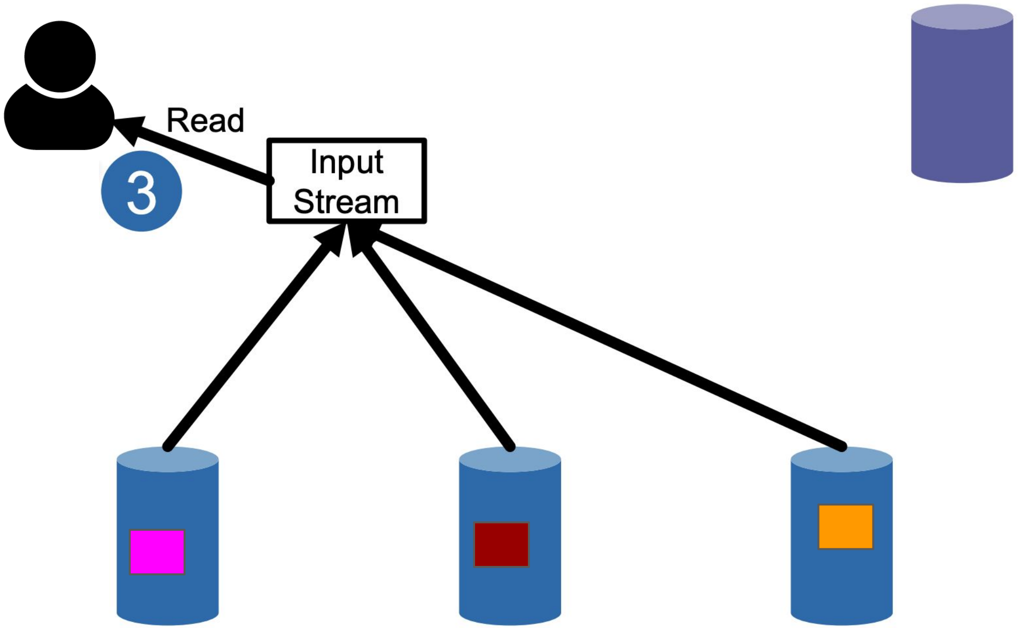
HDFS

- Detailed protocol: Read



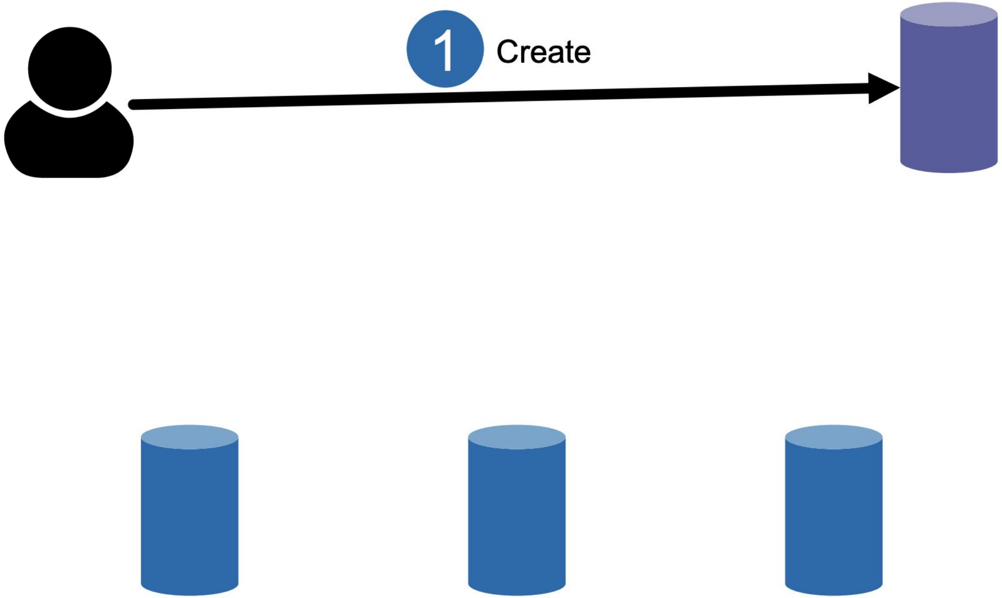
HDFS

- Detailed protocol: Read



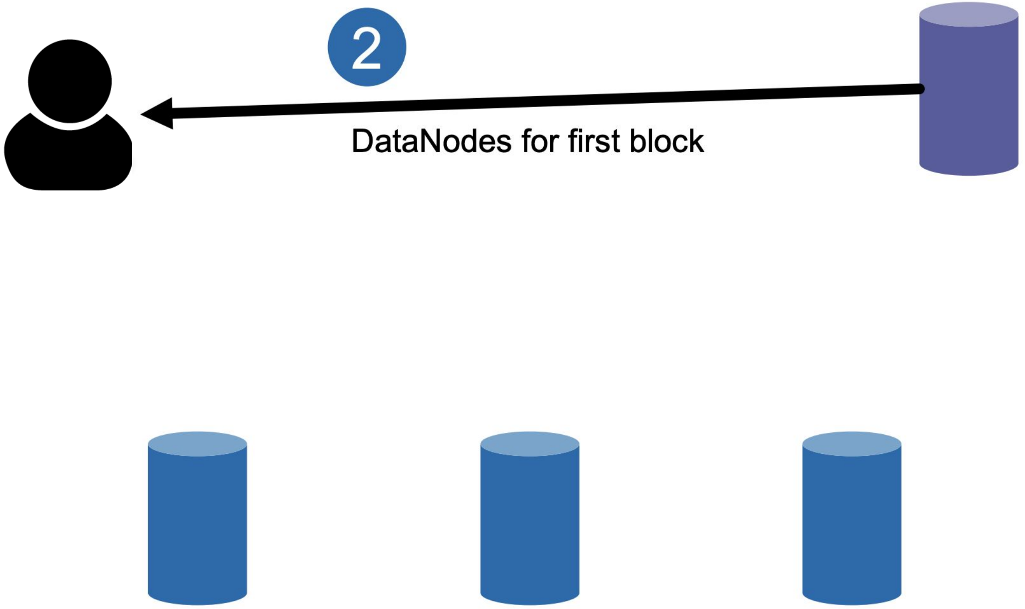
HDFS

- Detailed protocol: Write



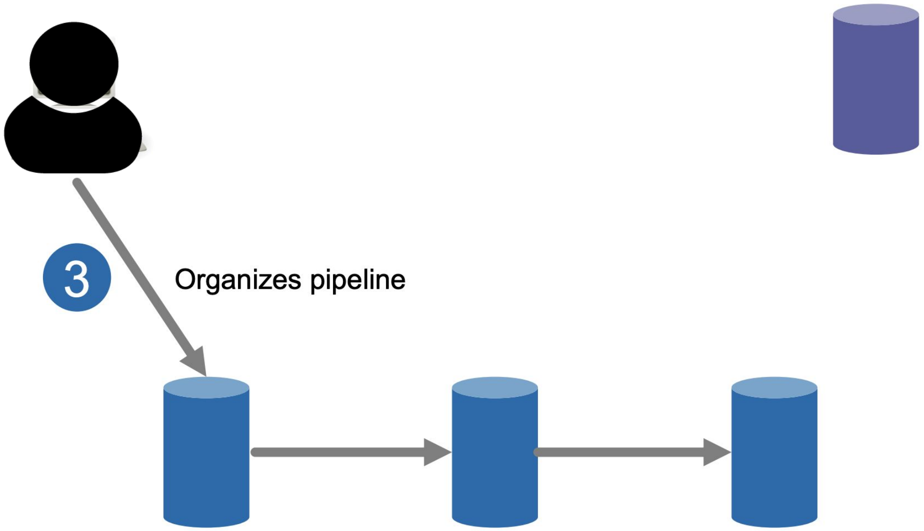
HDFS

- Detailed protocol: Write



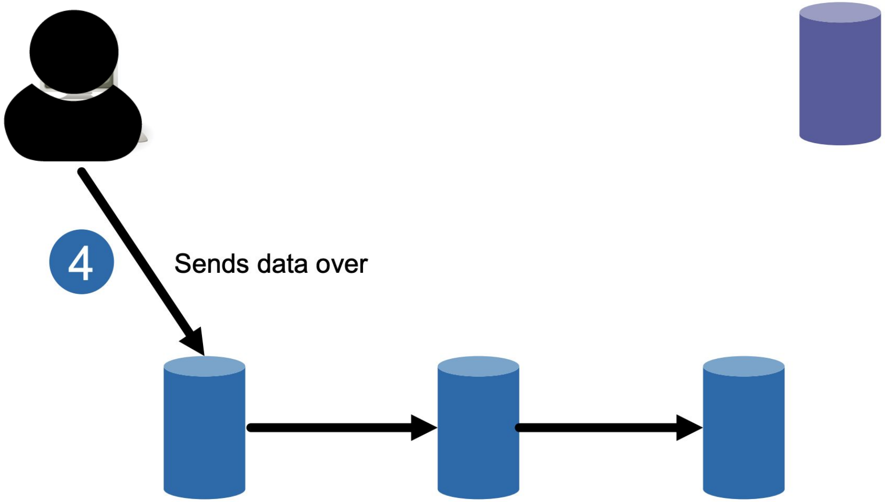
HDFS

- Detailed protocol: Write



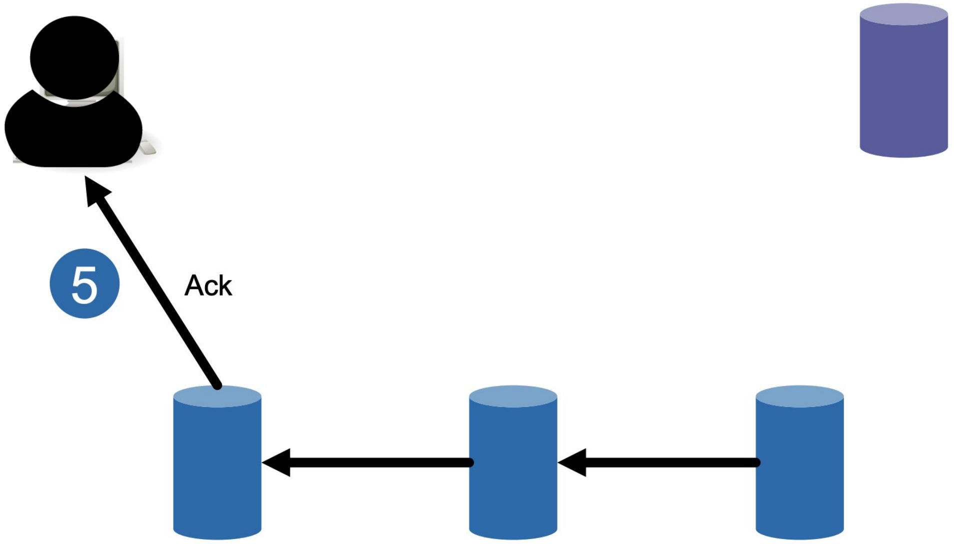
HDFS

- Detailed protocol: Write



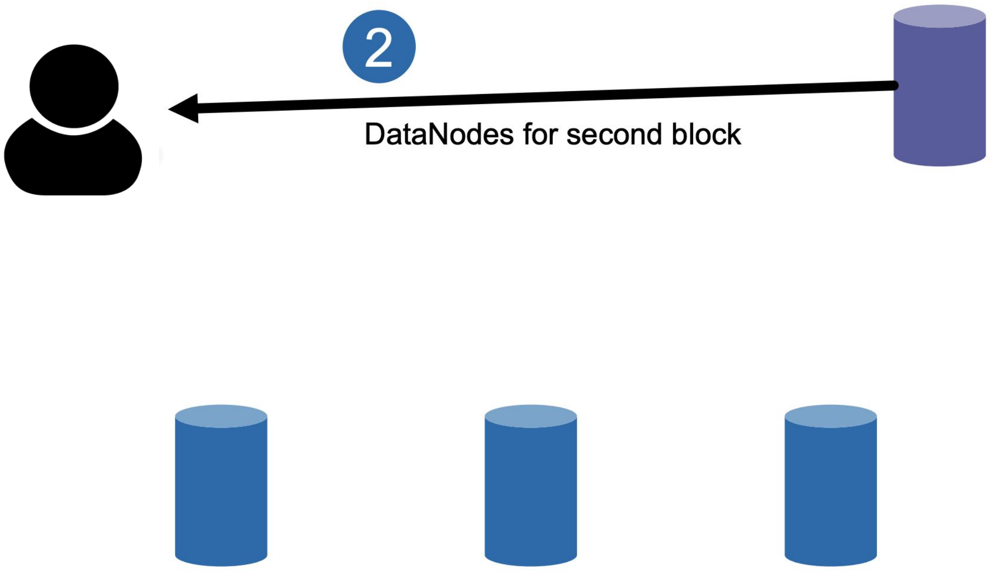
HDFS

- Detailed protocol: Write



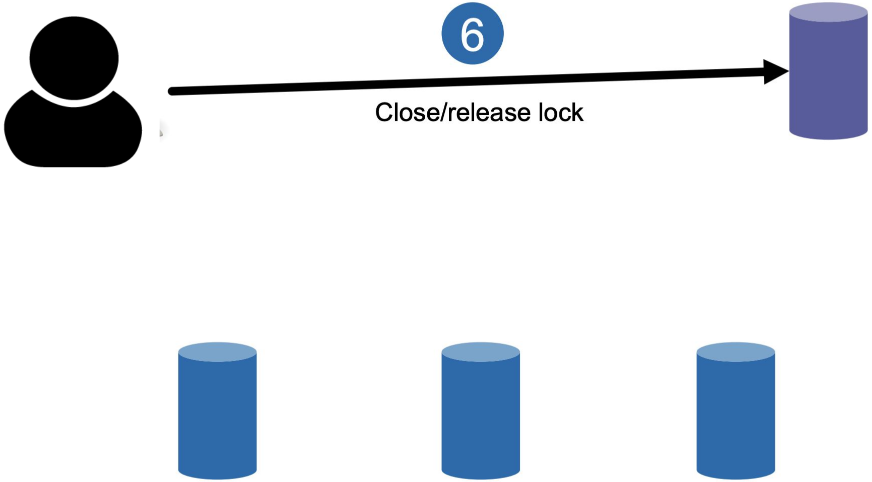
HDFS

- Detailed protocol: Write



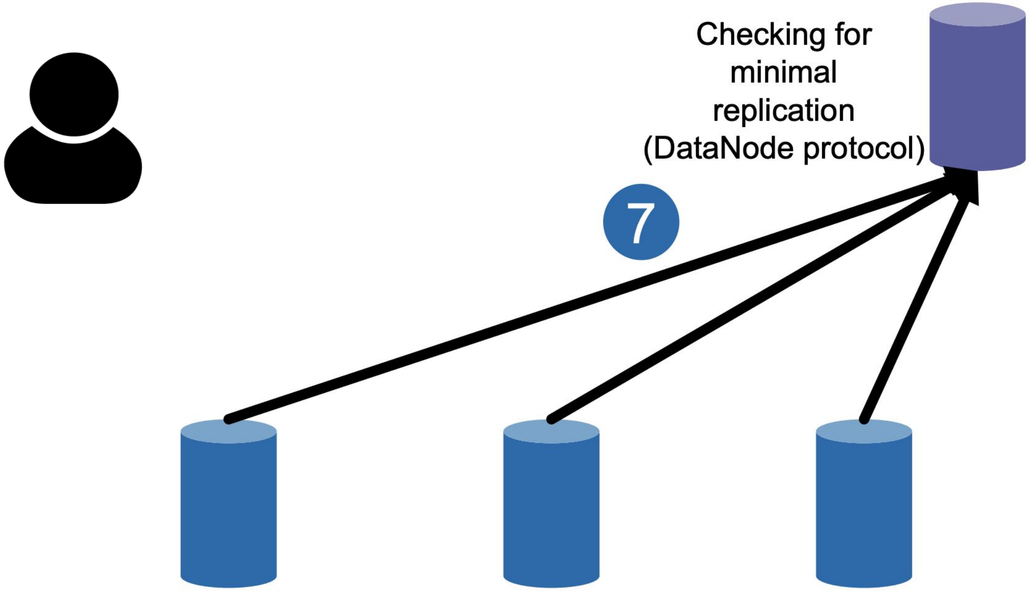
HDFS

- Detailed protocol: Write



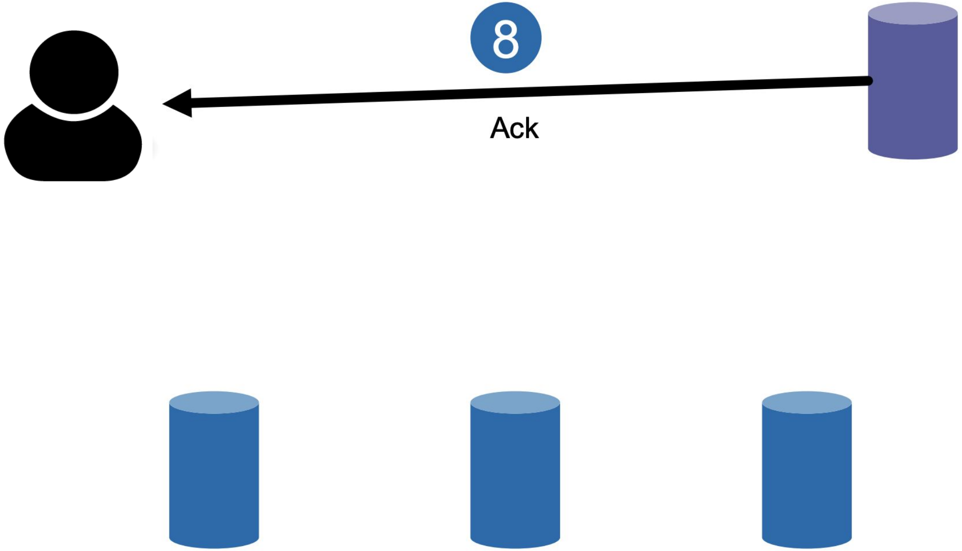
HDFS

- Detailed protocol: Write



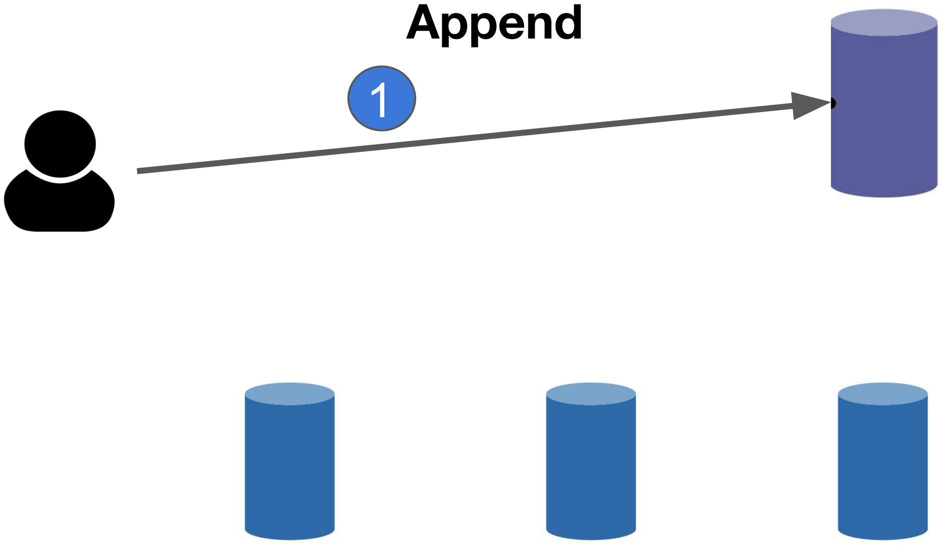
HDFS

- Detailed protocol: Write



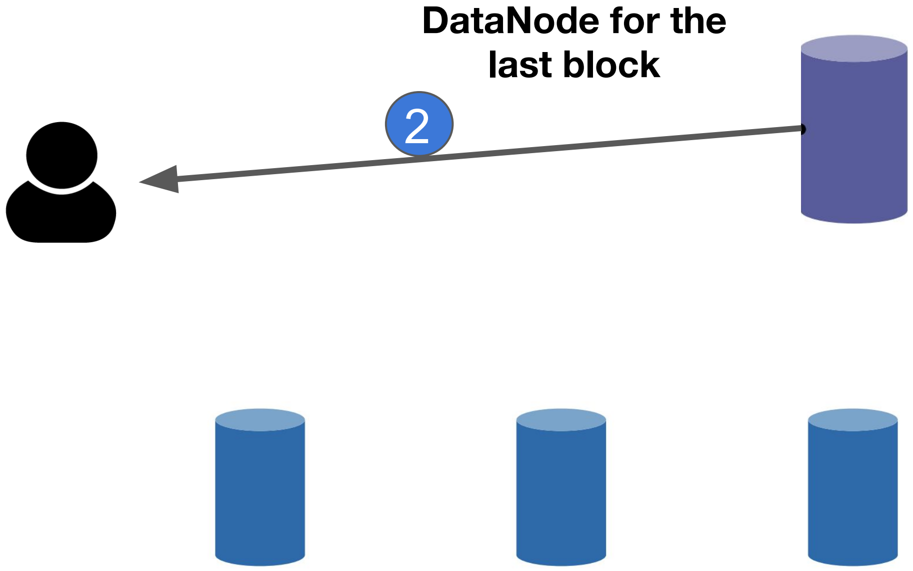
HDFS

- Detailed protocol: Append



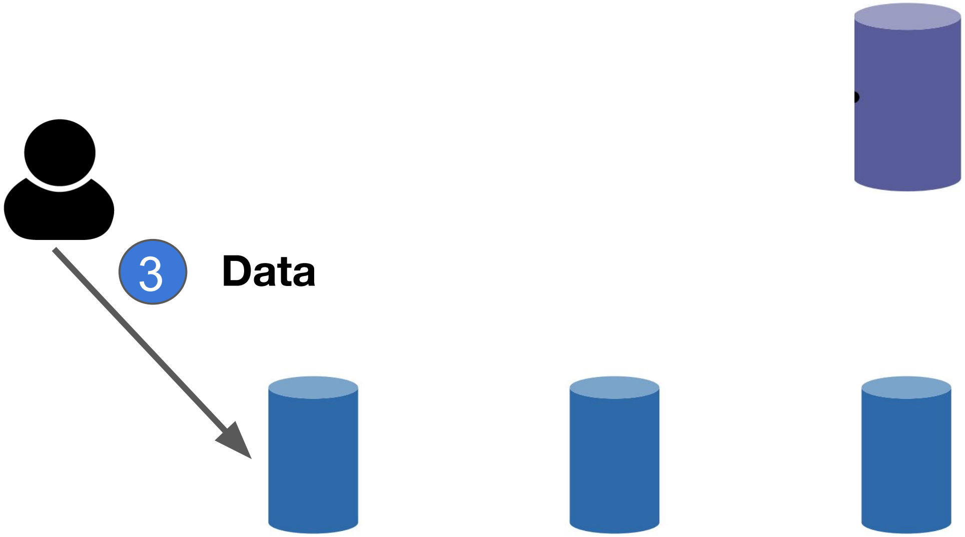
HDFS

- Detailed protocol: Append



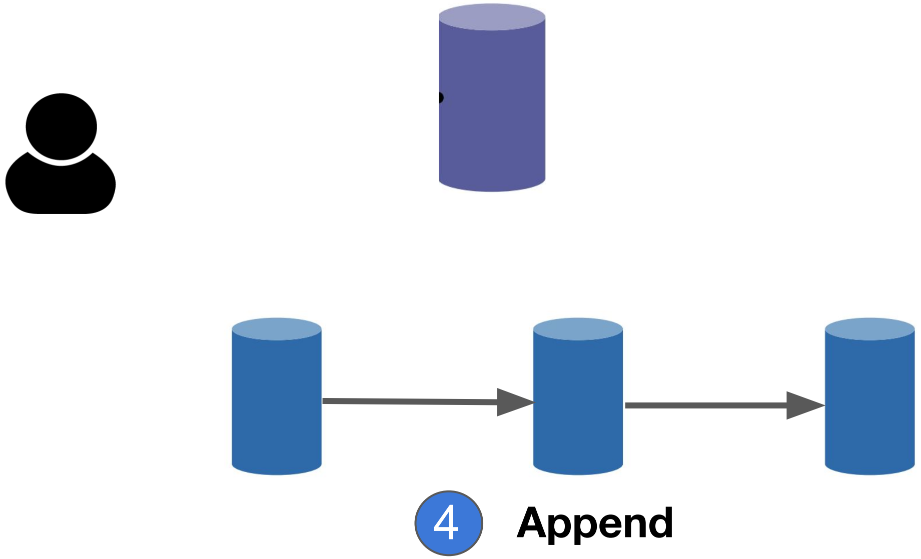
HDFS

- Detailed protocol: Append



HDFS

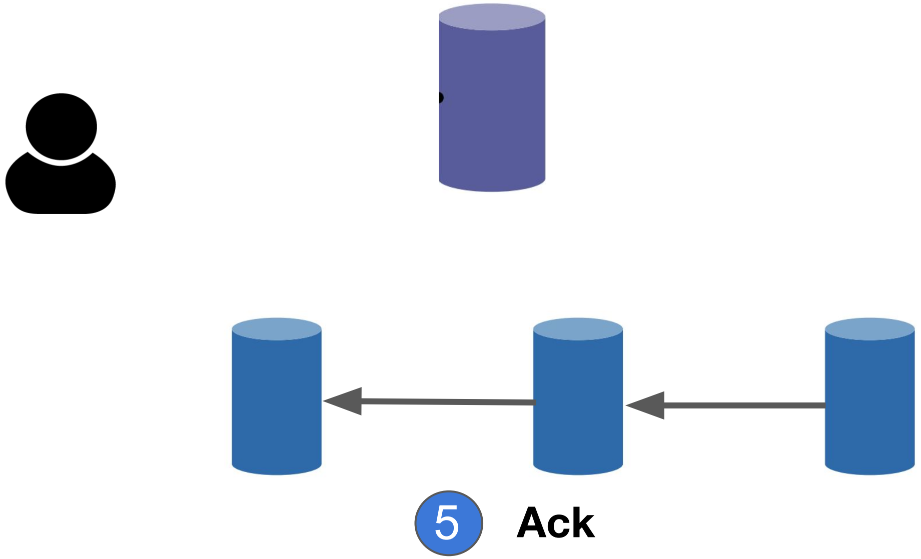
- Detailed protocol: Append



Each node will append data to the block

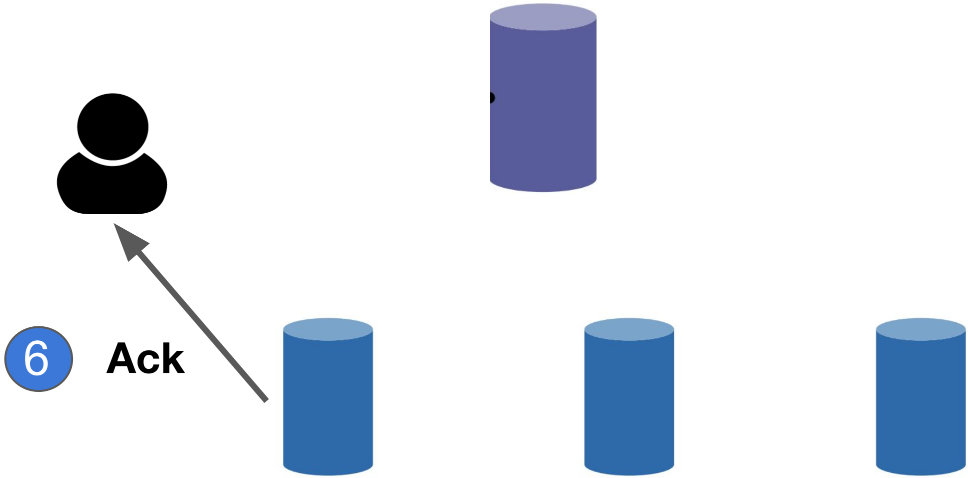
HDFS

- Detailed protocol: Append



HDFS

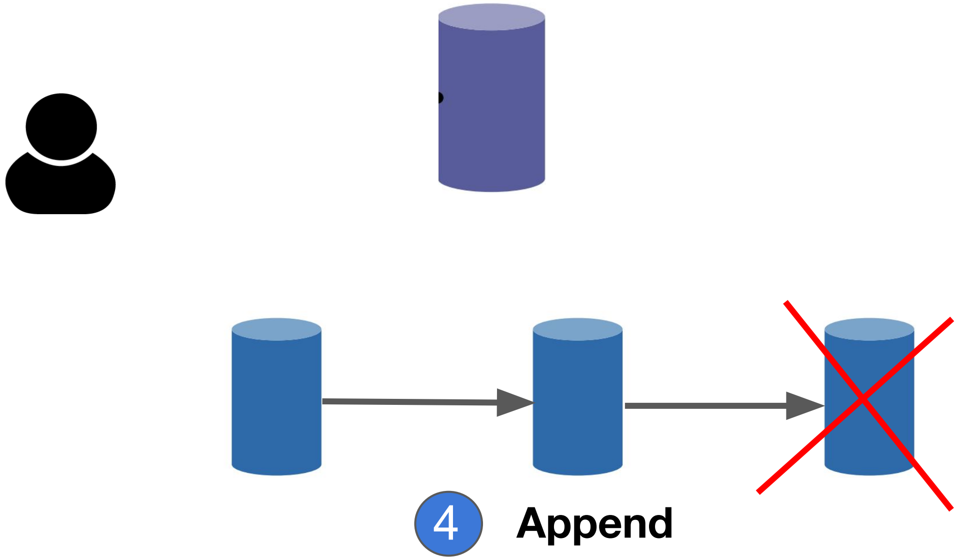
- Detailed protocol: Append



Only when all nodes append successfully, return OK to client

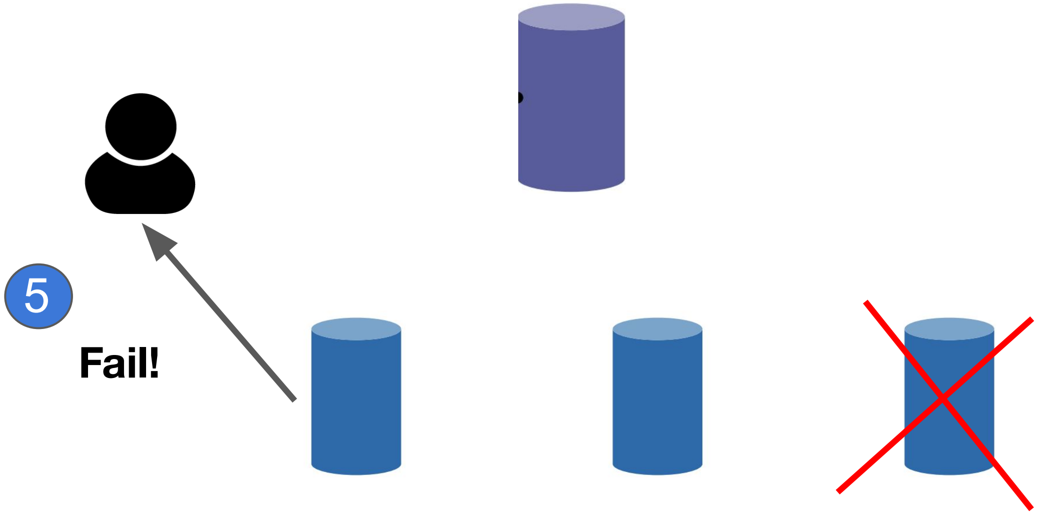
HDFS

- Detailed protocol: Append
  - If fail, client retries



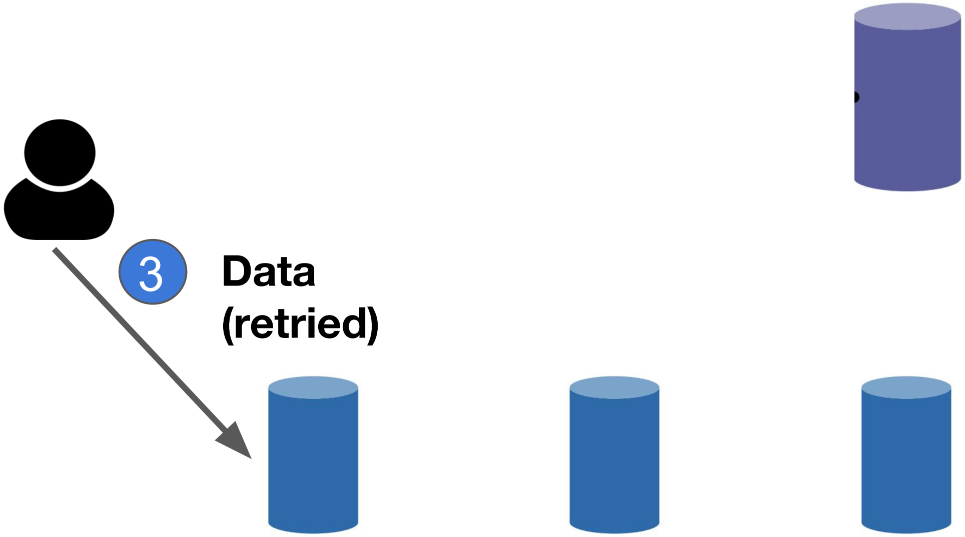
HDFS

- Detailed protocol: Append
  - If fail, client retries



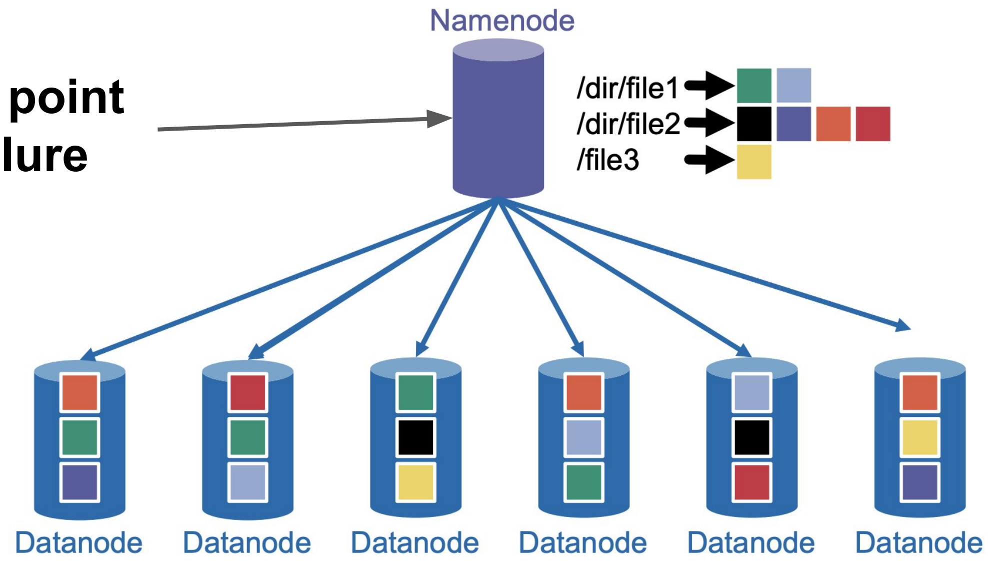
HDFS

- Detailed protocol: Append
  - If fail, client retries



HDFS

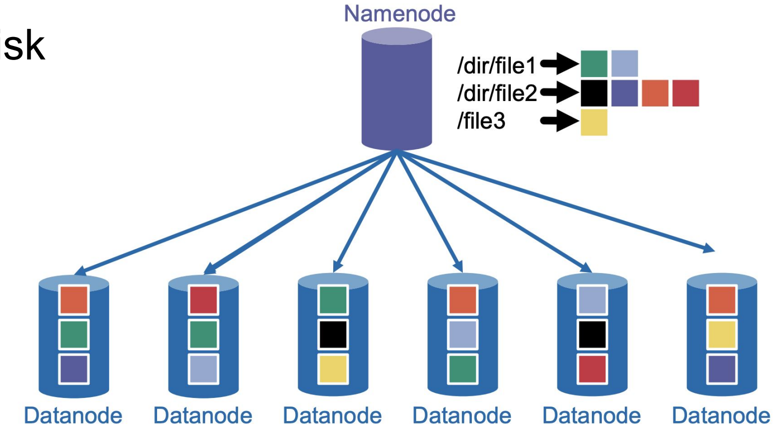
- One major problem:



Single point of failure

HDFS

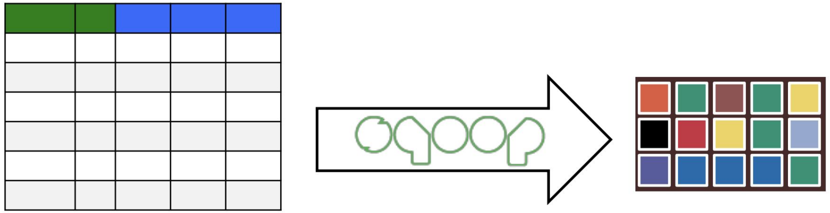
- What if it fails:
  - Approach 1: every request is a transaction
    - Then use WAL (Week 8) to persist to disk
    - If fail, reboot
    - Takes long time
  - Approach 2: keep hot standby
    - Better recovery
    - More expensive



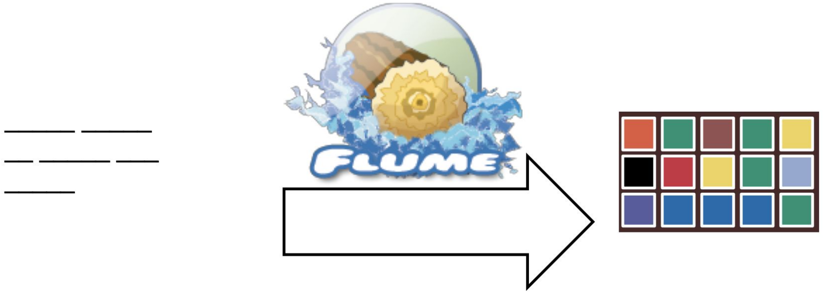
HDFS

- How to get data in:
  - HDFS shell commands
  - Tools!

Relational data → HDFS



Logs data → HDFS



Summary

- Google File System hugely influential
  - Scalable, fault-tolerant
  - Designed for specific workloads
- HDFS implements GFS
- HDFS de-facto distributed file system in the cloud
  - All cloud-based data analytics systems support reading from HDFS

Question 1

- Consider HDFS append operation
    - It doesn’t provide correctness!
    - Give an example of how incorrect append could happen.

Duplicates *even though they return successfully*
> append successfully to node --> fail this node
> client retries so it keeps appending to the first few nodes that were successful

Each record needs to give unique ID. Each time u append, you check the item before you, if contents same = duplicate so throw them. Needs to be handled in application layer

Question 2

- Consider HDFS append operation
  - Why do you think it’s difficult to guarantee correctness for append?

***Expensive** to make sure 3 replicas agree whether they have appended the value anot.*
If they havent -> can retry
If any one did not -> need to send data over to that node to replicate before you can retry