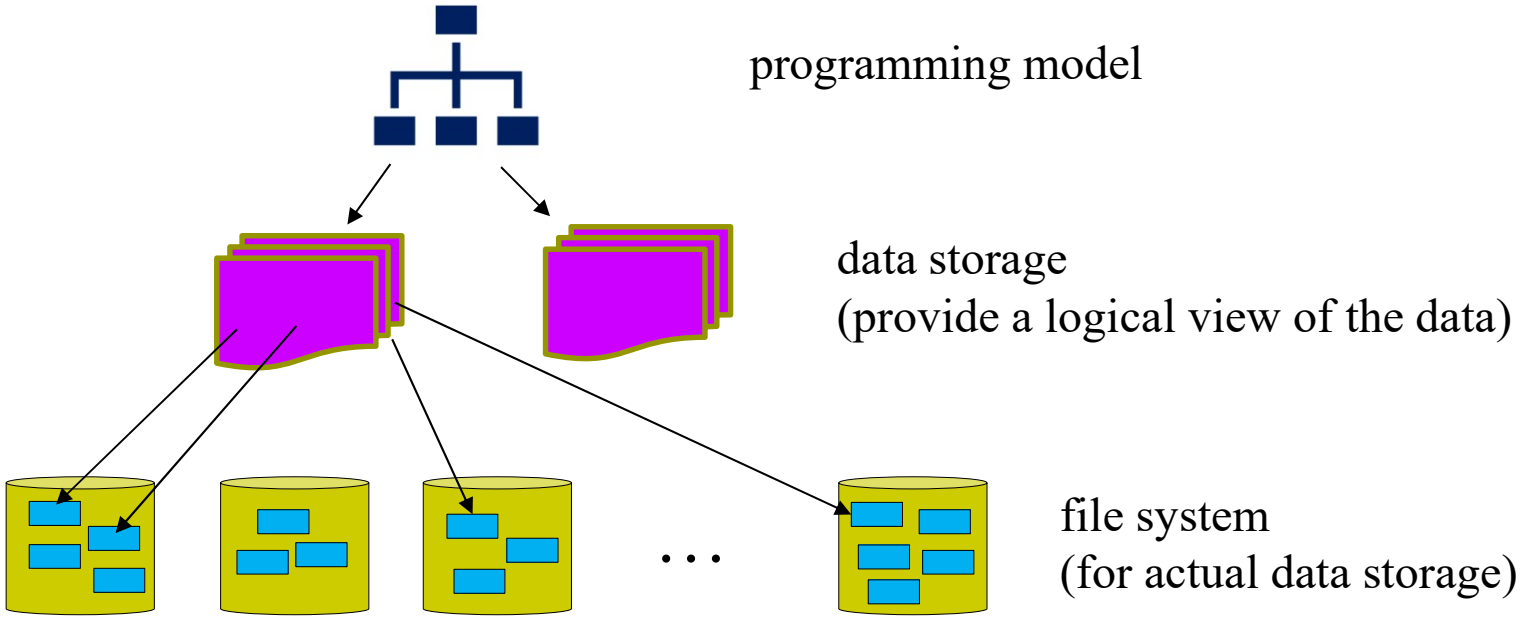
分散式系統雲端與雲端應用開發實務

Large Distributed File and Storage Systems

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

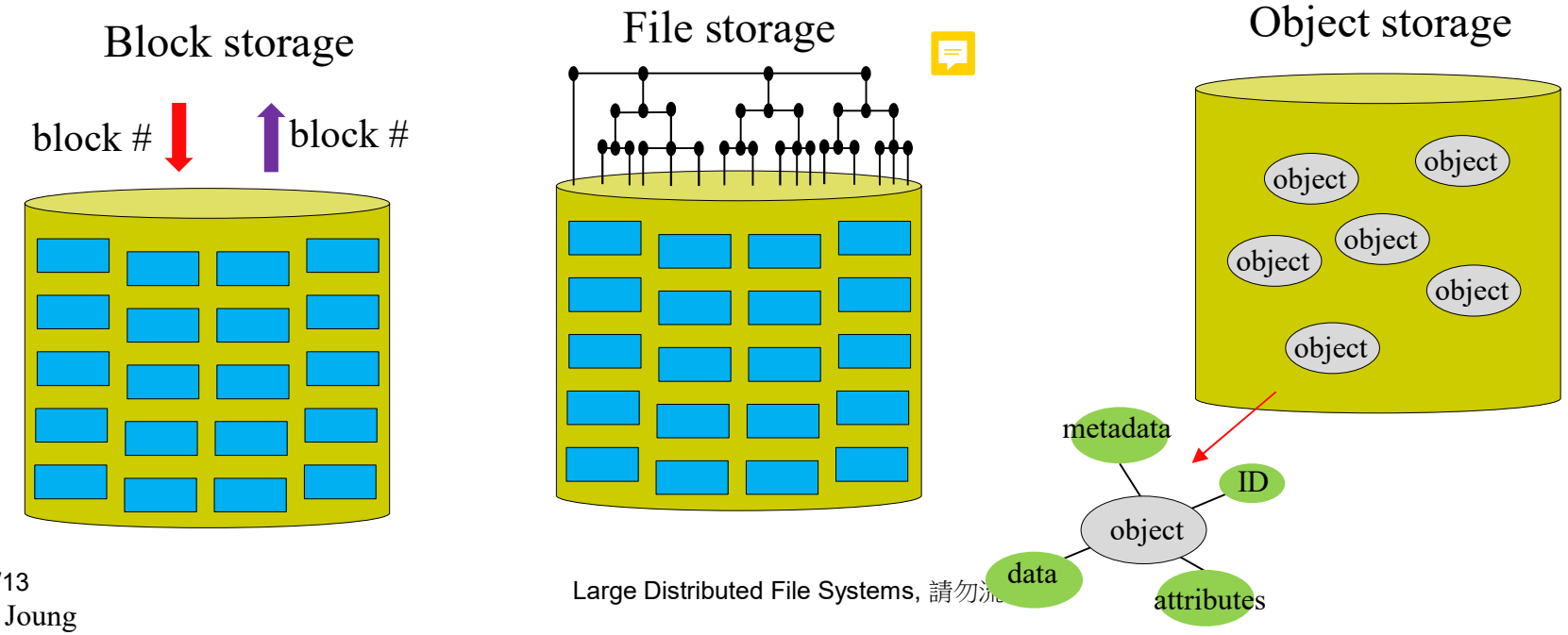
Building Bigdata Applications

- You need
  - a **programming model** for processing the data (e.g., MapReduce)
  - a **database-like storage system** for data read/write (e.g., Bigtable)
  - a **file system** to physically store the data (e.g., GFS)

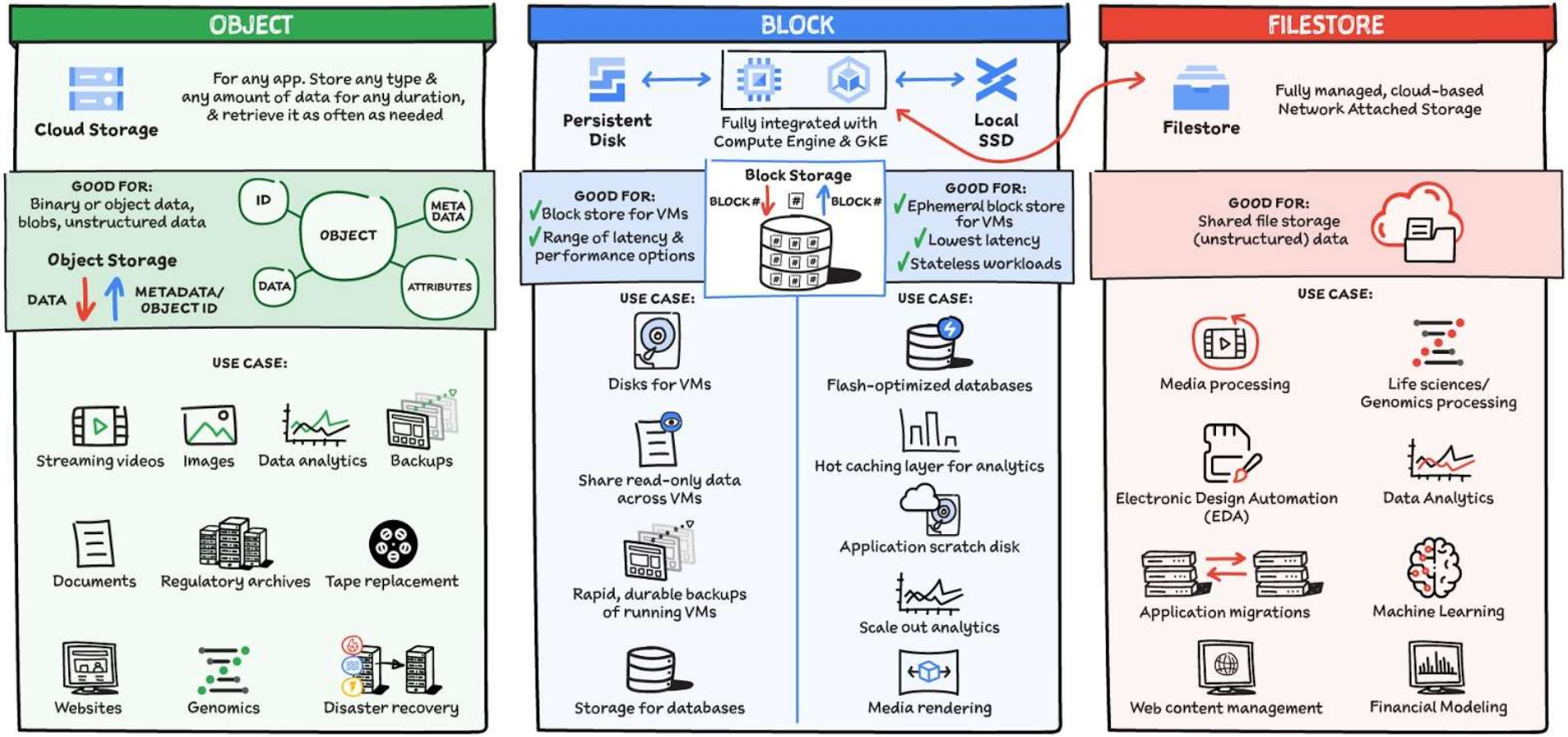


Object vs. Block vs. File storage

- **Block** storage: manage data as blocks within physical devices
- **File** storage: manage data as a file hierarchy
- **Object** storage: manage data as objects containing, in addition to its data, ID, metadata, and attributes
  - IDC estimated (2019) that unstructured data will represent 80% of all data worldwide by 2025
    - Emails, presentations, spreadsheets, photos, videos, audios, …



Example: Support in Google Cloud



Source: Google Cloud

File Systems vs. Database

- File System: a collection of raw data files.
- Databases:
  - although also used to store data, they are designed particularly for easily organizing, storing and retrieving large volume of data

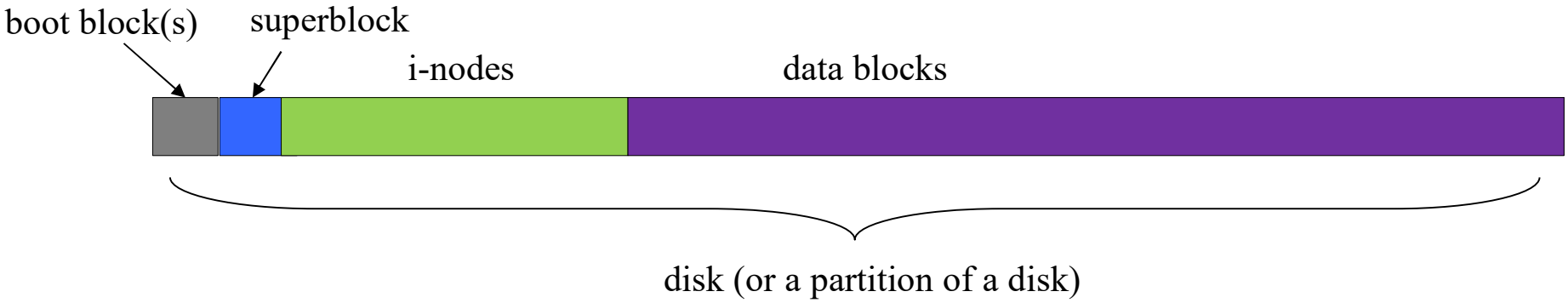
Three parts in this module

- Large distributed file systems
- Large distributed storage systems
- MapReduce programming model

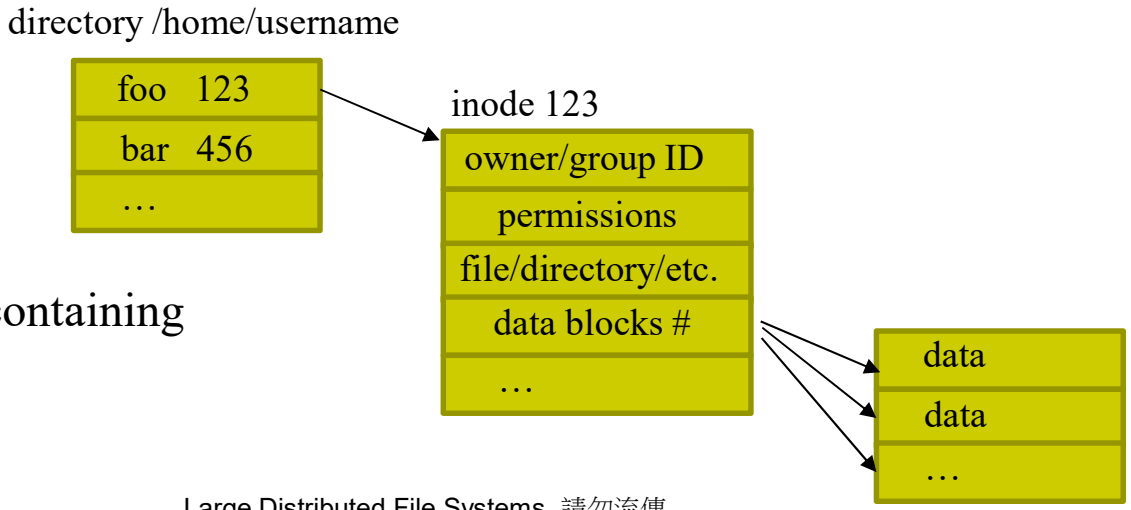
Centralized File Systems

- Unix/Linux
- DOS FAT

Unix-like File System

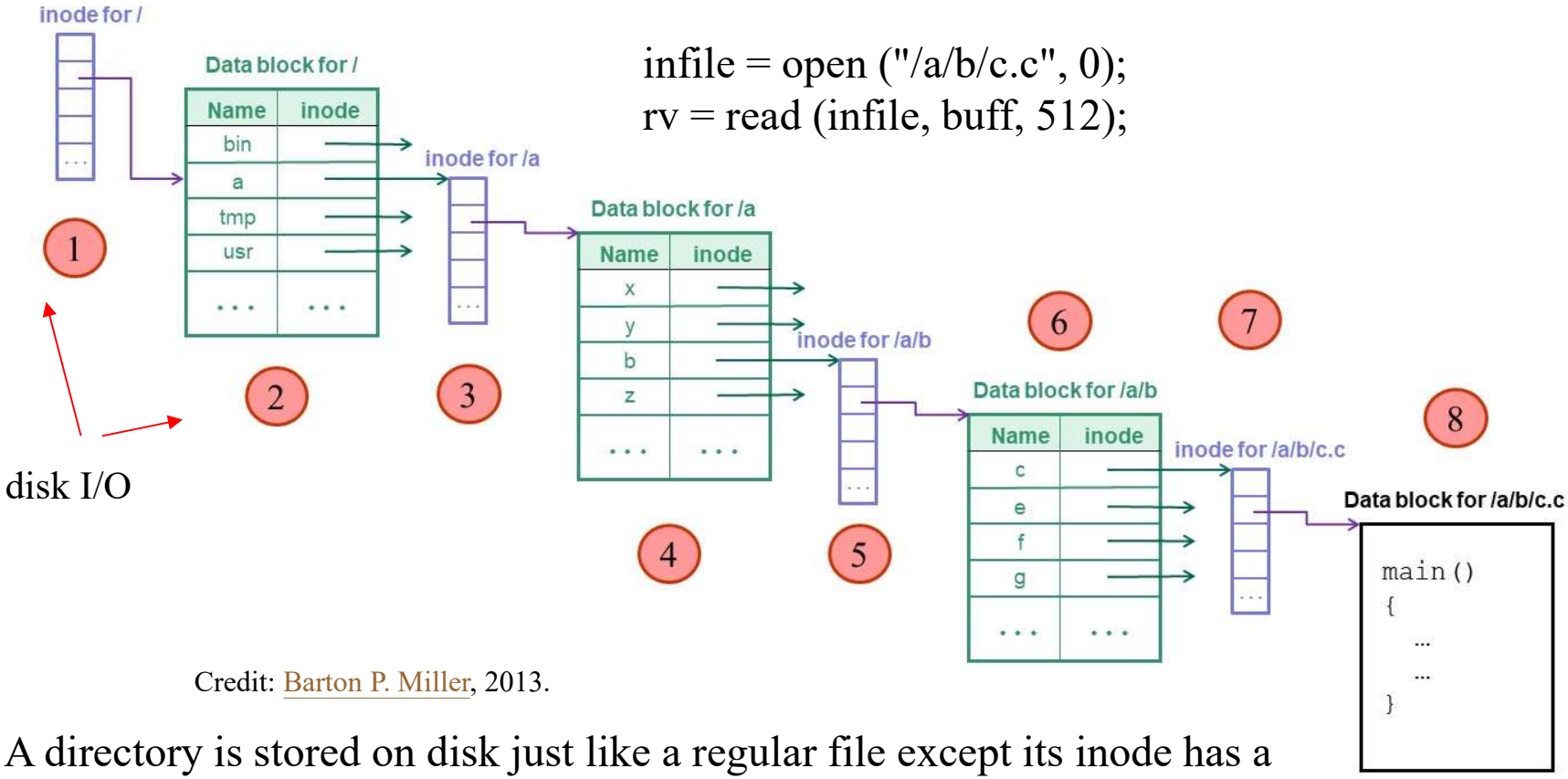


- boot block holds the loader to boot the operating system
- superblock contains basic information about the entire file system, e.g., size, the list of free and allocated blocks, partition name, modification time...
- each inode ("**i**nformation **nodes**") holds a pointer to the disk blocks containing the data in the file



A directory is just a file containing <name, i-number> pairs.

Unix File System: An Illustration



Credit: Barton P. Miller, 2013.

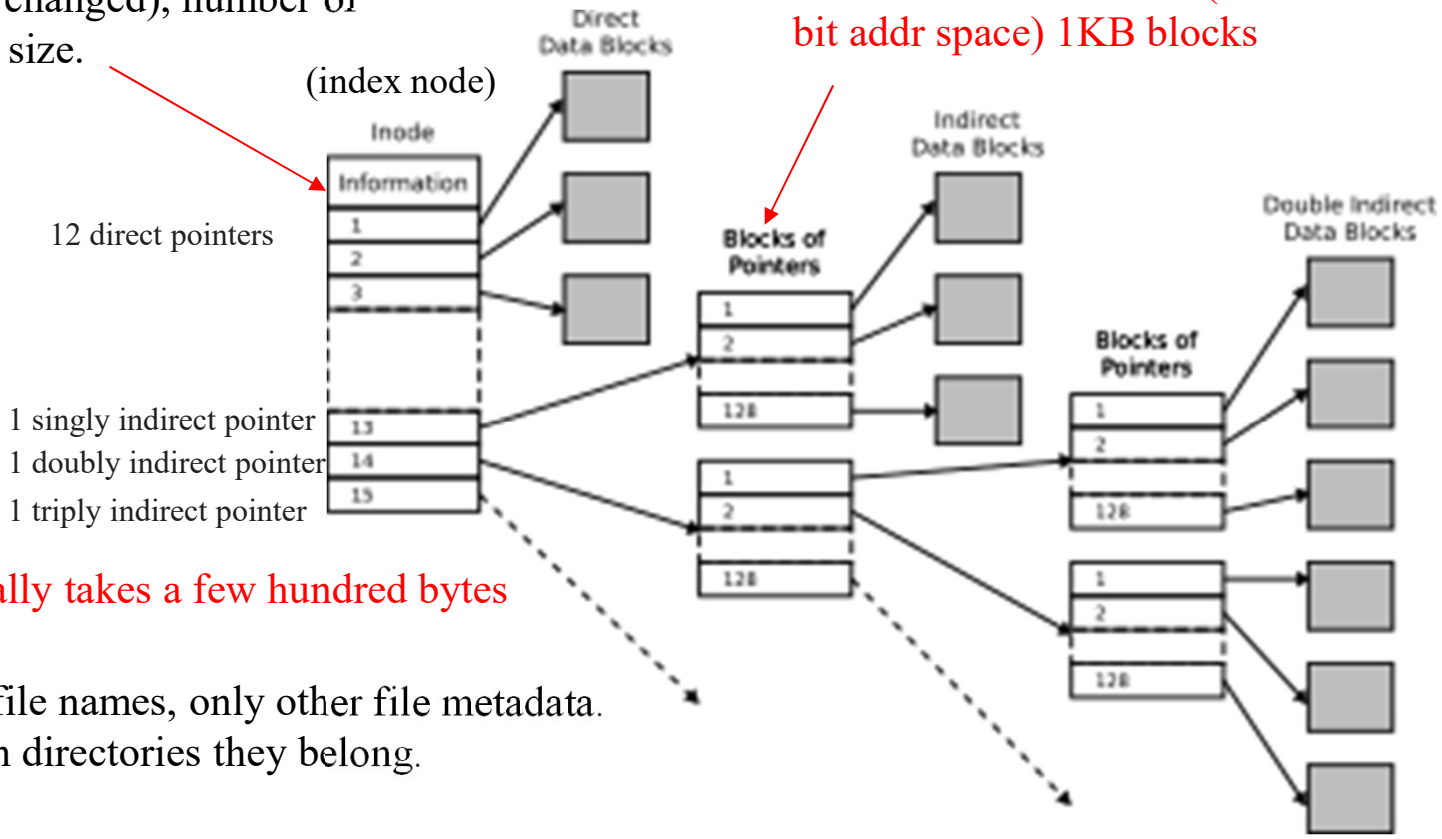
- A directory is stored on disk just like a regular file except its inode has a special flag bit set to indicate that it's a directory.
- Each directory contains <name, i-number> pairs in no particular order.
- Root directory has no name, and has i-number 2 (inode 1 is used to keep track of bad blocks on the disk, and there is no inode 0, as '0' is used as a NULL value to indicate that there is no inode.)

Unix i-node

File's metadata: file owner, group owner, file type, access permissions, timestamps (create, last access, and last changed), number of links to the file, file size.

block size are typically 512B, 1KB or 4 KB

A 4KB block can index (in a 32-bit addr space) 1KB blocks



inode typically takes a few hundred bytes

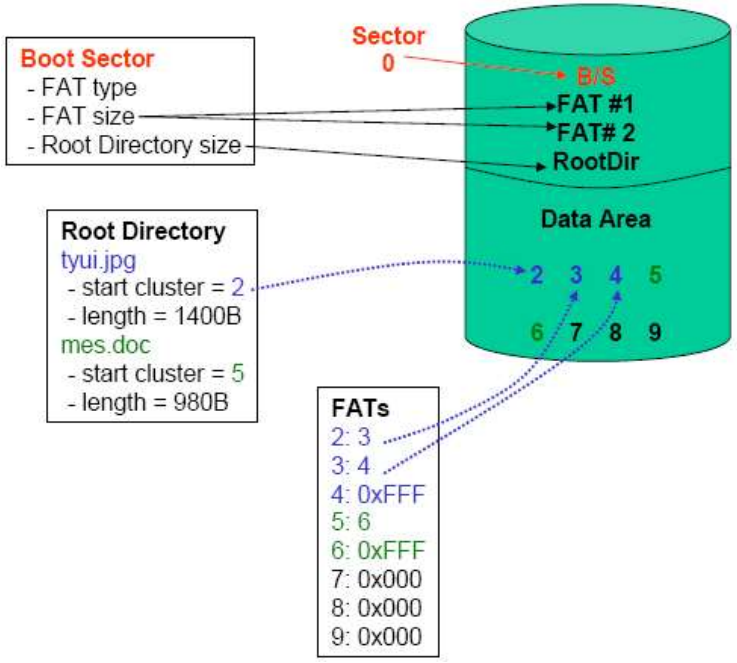
inodes do not contain file names, only other file metadata. Filenames are stored in directories they belong.

For a 32-bit system, each pointer takes 4 bytes. Then, what is the max. file size can be supported? Ans. 4TB.

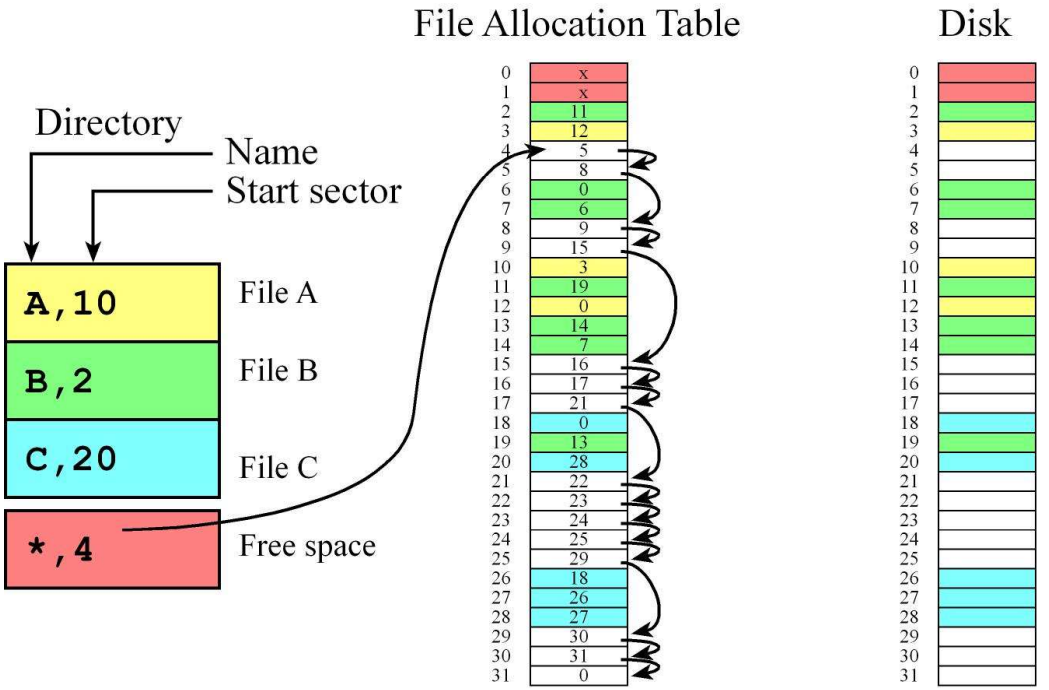
Source: wiki

File Allocation Table (FAT)

A file system designed in 70's for MS-DOS and Windows



Source: http://www.c-jump.com/CIS24/Slides/FAT/FAT.html



Source: https://github.com/monpeco/real_time_bn/blob/master/lab5/File_systems.md

What Can Be Learned when Designing a Large Distributed File System?

- We need a similar index structure for data blocks
  - but who maintain it and how?
- How to scale?
- How to cope with failures?
  - Availability
  - Semantics
    - Will read operation return the latest write data?
    - This affects:
      - Cache Consistency
      - Writing Policy
        - Write Through vs. Delayed Write

Goal

- Design a Distributed File System that is
  - Scalable
  - Reliable/Fault-Tolerant
  - with High-Performance
- Idea: to place data blocks across different hosts and focus on how to manage the metadata (index/structural information) of the data blocks

Large Distributed File Systems Case Studies

- HDFS - Hadoop Distributed File System (HDFS)
- The Google File System (GFS)
- Ceph Distributed File System

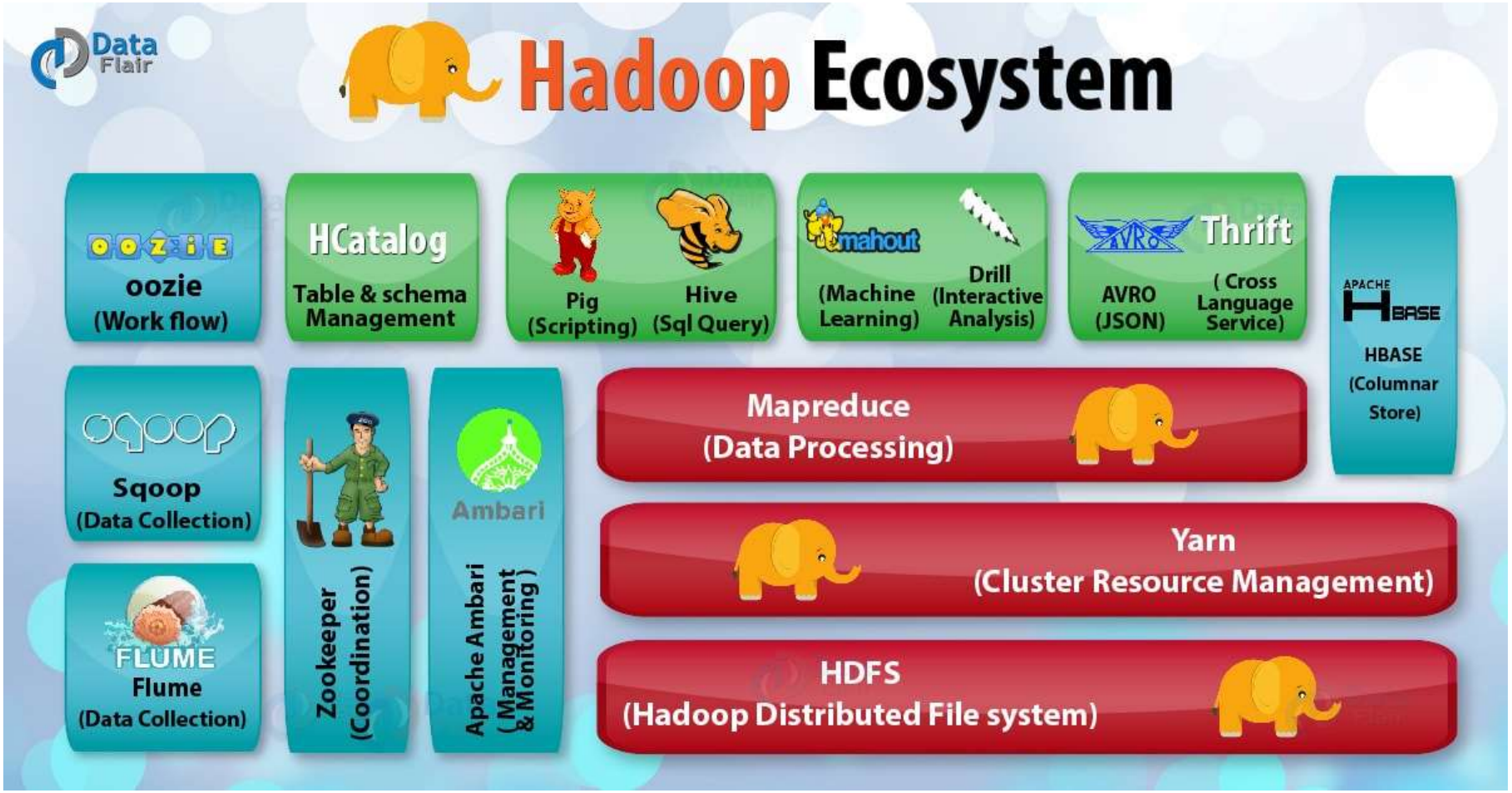
(Apache) Hadoop

- **Reading**
  - The Hadoop distributed file system: Architecture and design, D. Borthakur, 2007.
    - Overall architecture description
    - Also available in
      - https://Hadoop.apache.org/docs/r1.2.1/hdfs_design.html
  - The Hadoop Distributed File System, K. Shvachko, H. Kuang, S Radia, R. Chansler, IEEE MSST 2010
    - Has more implementation details used by Yahoo!

(Apache) Hadoop

- Apache top level project, open-source software for reliable, scalable, distributed computing and data storage
- Apache Hadoop software library is a framework that allows for the **distributed processing of large data sets** (**bigdata**) across clusters of computers using simple programming model
  - Very suitable for the **MapReduce** paradigm
- Designed to scale up from single servers to thousands of machines, each offering local computation and storage
- Designed to detect and handle failures at the application layer

Hadoop Ecosystem/Stack



Source: https://data-flair.training/blogs/hadoop-ecosystem-components/

ps. There are a number of views of Hadoop ecosystem.

Developed History

- 2005: Doug Cutting and Michael J. Cafarella developed Hadoop to support distribution for the Apache Nutch search engine project.
- The project was funded by Yahoo.
- 2006: Yahoo gave the project to Apache Software Foundation
- Hadoop ecosystem was mostly written in Java

HDFS Design Goals

- A highly fault-tolerant, high throughput distributed file system designed to run on commodity hardware (typically run a GNU/Linux OS)
  - built using the Java language for portability
  - An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data.
  - supports a traditional hierarchical file organization
- Not general purpose, designed more for batch processing rather than interactive use by users

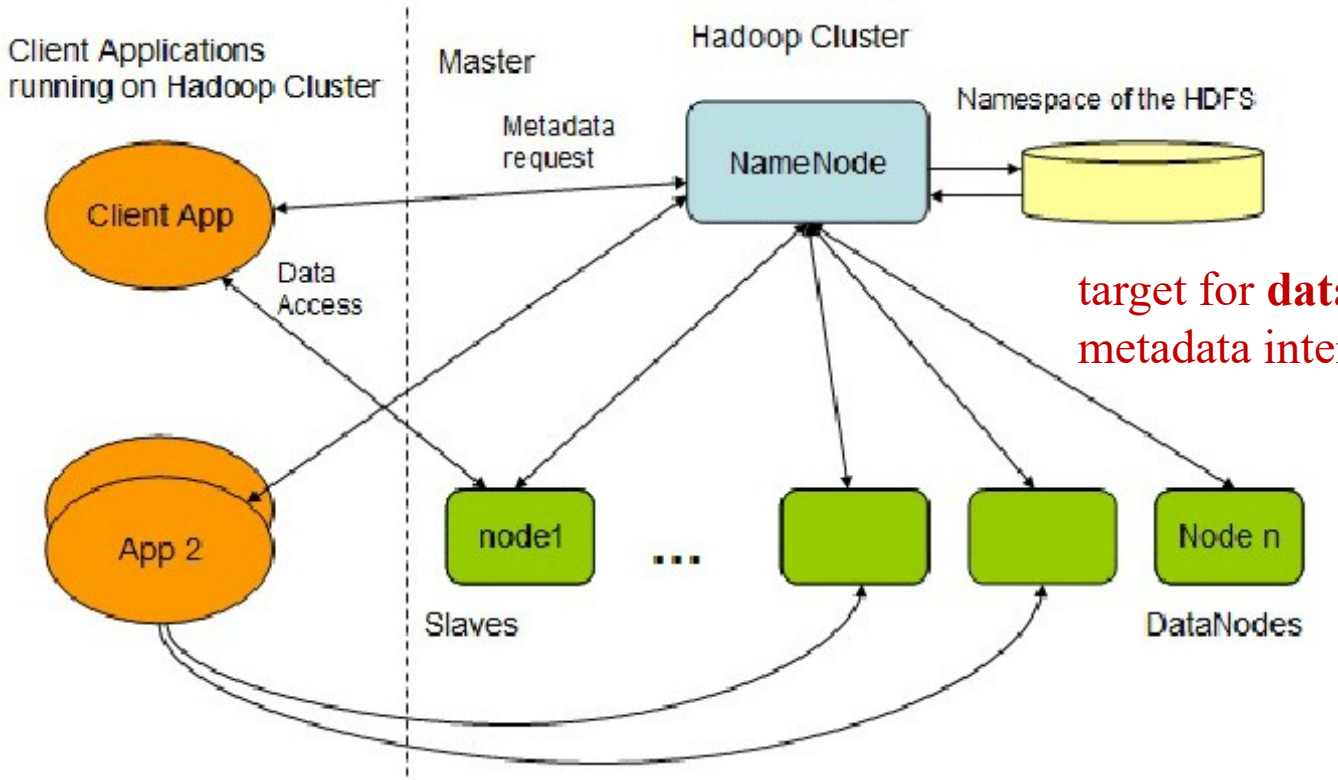
Note that HDFS (published in 2007) was built on the design of Google's GFS (published in 2003).

Hadoop Design Goals (2)

- Emphasizing on high throughput of data access rather than low latency of data access.
  - Large Data Sets: a typical file in HDFS is gigabytes to terabytes in size, with tens of millions of files in a single instance.
    - Because data size is large, moving computation is cheaper than moving data
    - HDFS provides interfaces for applications to move themselves closer to where the data is located.
  - Assume a **write-once-read-many** access model for files
    - simplifies data coherency issues and enables high throughput data access.
    - Applications: MapReduce, web crawler
    - Future extension: appending-writes to files.

Master/Slave Architecture

- An HDFS cluster consists of a single **Namenode** and a number of **Datanodes.**



target for **data intensive** rather than metadata intensive applications

Source: https://www.researchgate.net/figure/Hadoop-Distributed-File-System-HDFS-Architecture_fig2_265403224

NameNode

- A master server that manages and maintain the filesystem namespace and regulates access to files by clients.
  - executes file system namespace operations like opening, closing, and renaming files and directories
  - determines the mapping of blocks to DataNodes.
  - the arbitrator and repository for all HDFS metadata

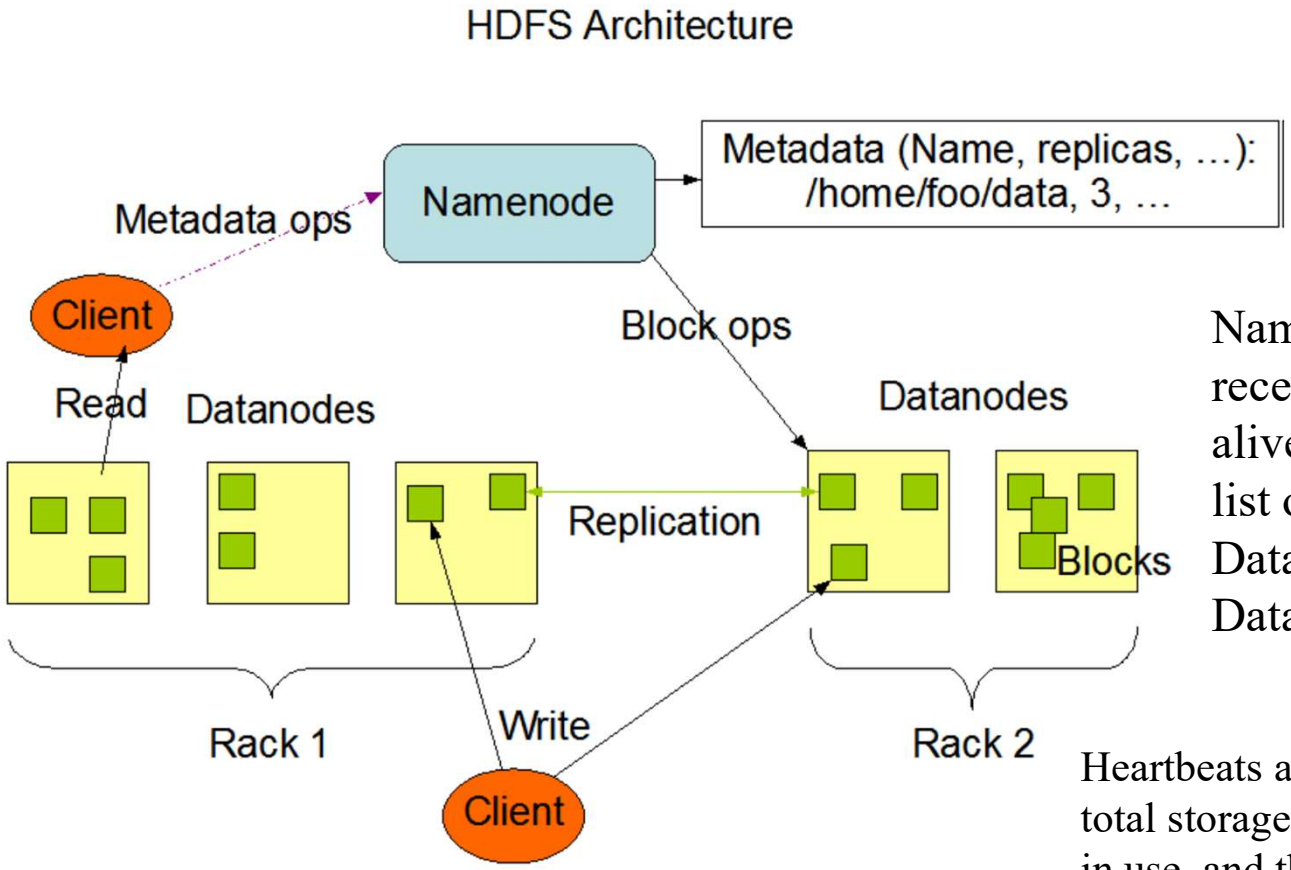
DataNode

- usually one per node in the cluster, which manages storage attached to the nodes that they run on.
  - Internally, a file is split into one or more blocks stored in a set of DataNodes.
- responsible for serving read and write requests from the file system's clients
- perform block creation, deletion, and replication upon instruction from the NameNode.
- During startup each DataNode connects to the NameNode and performs handshaking to verify the namespace ID and the software version of the DataNode, and then registers with the NameNode.

Data Replication

- HDFS stores each file as a sequence of blocks; all blocks in a file except the last one have the same size.
- The blocks of a file are replicated for fault tolerance.
  - The block size and **replication factor** are configurable per file.
  - The replication factor can be specified at file creation time and can be changed later.

Hadoop Architecture (2)

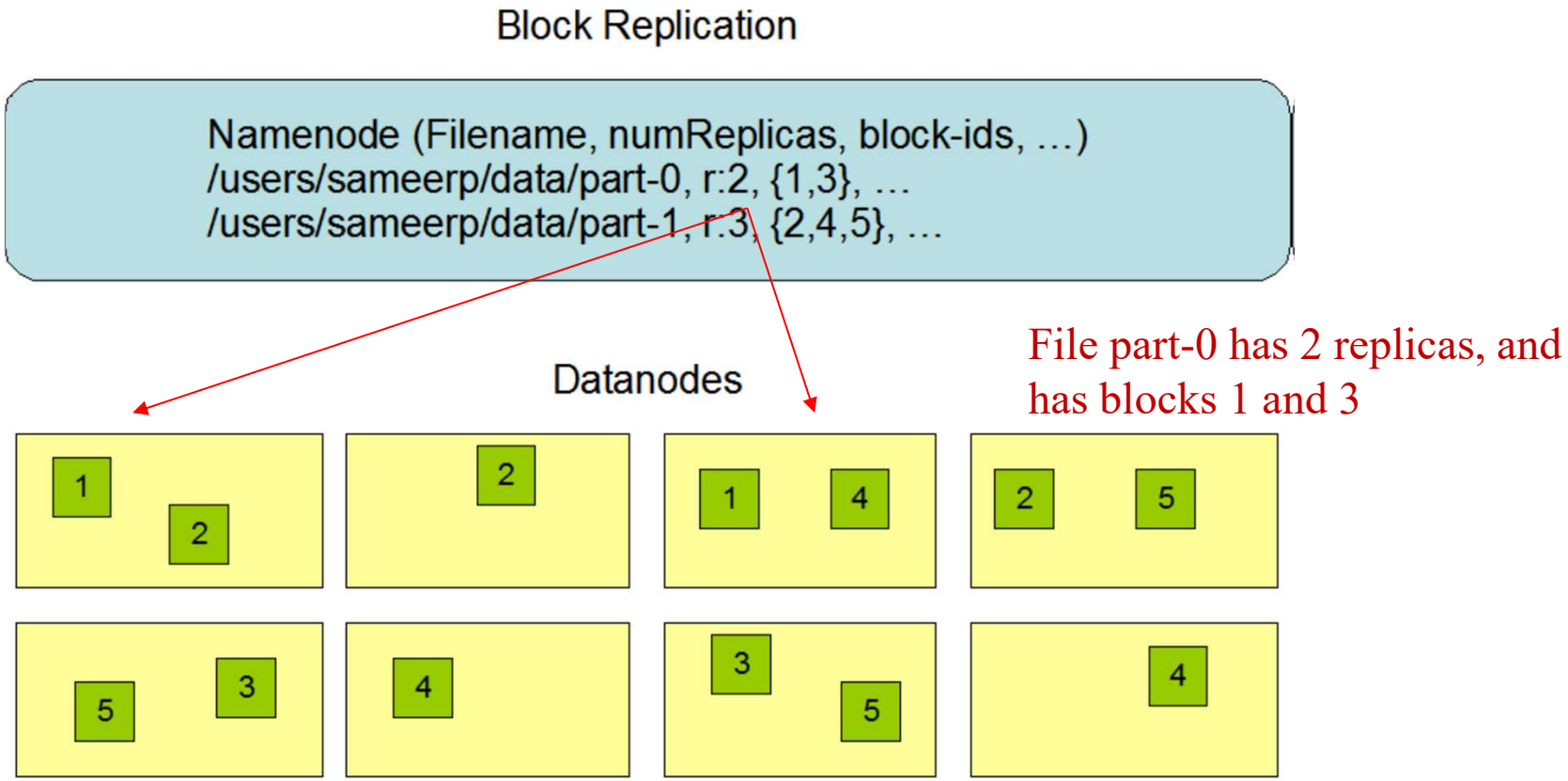


Namenode periodically receives a **Heartbeat** ("I am alive") and a **Blockreport** (a list of all blocks on a DataNode) from each of the DataNodes in the cluster.

Heartbeats also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's space allocation and load balancing decisions.

source: The hadoop distributed file system: Architecture and design, D. Borthakur, 2007

Block Replication



source: The hadoop distributed file system: Architecture and design, D. Borthakur, 2007

Additional Roles of NameNode

- Primary role: serving client requests
- Additional roles:
  - CheckpointNode
    - periodically combines the existing checkpoint and journal (EditLog) to create a new checkpoint and an empty journal.
  - BackupNode
    - Creating periodic checkpoints, but in addition it maintains an in-memory, up-to-date image of the file system namespace that is always synchronized with the state of the NameNode.

Replica Placement

- **rack-aware replica placement policy:**
  - to improve data reliability, availability, and network bandwidth utilization
    - Each NameNode determines the rack id each DataNode belongs to via the process outlined in Hadoop Rack Awareness on startup
  - Example: replication factor 3 (default)
    - put one replica on one node in the local rack,
    - another on a node in a different (remote) rack,
    - and the last on a different node in the same remote rack (??)
    - Additional replicas are randomly placed
  - needs lots of tuning and experience
  - Replica Selection
    - Clients read from nearest replica

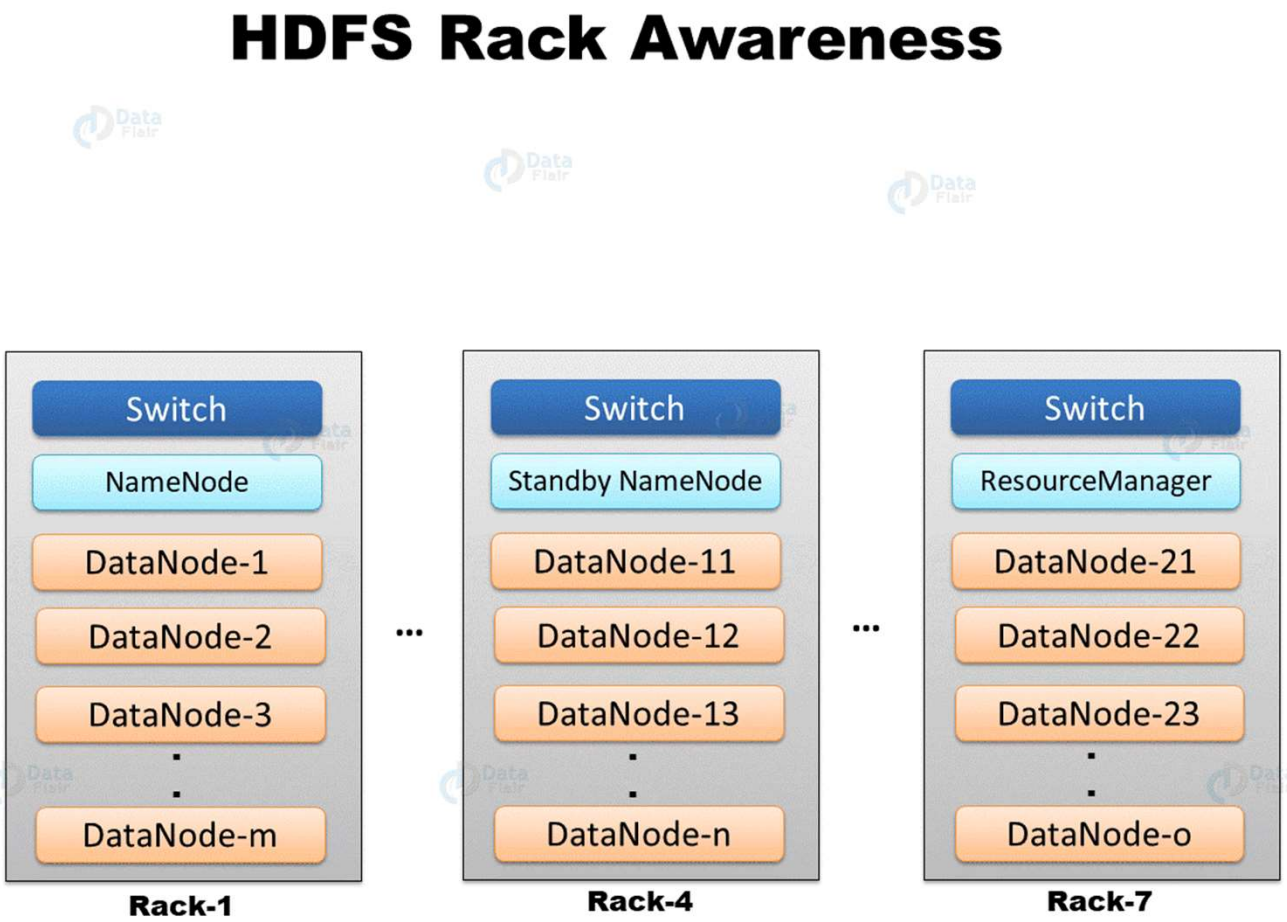
More about Replica Placement

For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a different node in the local rack, and the last on a different node in a different rack. (The Hadoop Distributed File System Architecture and design, 2007)

For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on one node in the local rack, another on a node in a different (remote) rack, and the last on a different node in the same remote rack. (HDFS Architecture Guide, Hadoop 1.2.1 Documentation)

When a new block is created, HDFS places the first replica on the node where the writer is located, the second and the third replicas on two different nodes in a different rack, and the rest are placed on random nodes with restrictions that no more than one replica is placed at one node and no more than two replicas are placed in the same rack when the number of replicas is less than twice the number of racks (The Hadoop Distributed File System, IEEE MSST 2010)

Rack Awareness Illustration



source: https://data-flair.training/blogs/rack-awareness-hadoop-hdfs/

place data close to the client, then rack-aware

Safemode

- On startup, the NameNode enters a special state called Safemode:
  - The NameNode receives Heartbeat and Blockreport messages from the DataNodes.
  - Each block has a specified minimum number of replicas.
  - A block is considered safely replicated when the minimum number of replicas of that data block has checked in.
  - After a configurable percentage of safely replicated data blocks checks in, the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas. The NameNode then replicates these blocks to other DataNodes.

HDFS Namespace

- The NameNode uses a transaction log called the `EditLog` (some implementations called "journal") to persistently record every change to file system metadata.
  - EditLog is stored in a file in the NameNode's local host OS file system.
  - Checkpoint can be applied to truncate `EditLog` when updates have been applied persistently to `FsImage`.
- The entire file system namespace, including the mapping of blocks to files and file system properties, is stored as a file called `FsImage` in the NameNode's local file system too.
  - The NameNode keeps an image of the entire file system namespace and file Blockmap in memory (4 GB RAM should be enough to support a huge number of files and directories)

HDFS Data Blocks

- DataNode stores HDFS data in files in its local file system.
  - Each block of HDFS data is stored in a separate file in the DataNode's local file system.
    - A typical block size used by HDFS is 64 MB or 128MB
  - DataNode has no knowledge about HDFS files.
  - It may not be optimal to create all local files in the same directory, but rather a hierarchical directory
  - When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files and sends this report (called Blockreport) to the NameNode.

Communication Protocols

- The All HDFS communication protocols are layered on top of the TCP/IP protocol.
- Client establishes a connection to a configurable TCP port on the NameNode machine using the ClientProtocol.
- DataNodes talk to the NameNode using the DataNode Protocol.
- A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNode Protocol.

Robustness

- HDFS handles three common types of failures: NameNode failures, DataNode failures and network partitions.
  - DataNode failures and network partitions are detected by absence of Heartbeat messages that are sent from DataNode to the NameNode periodically.
  - The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary.
  - Block corruption is implemented by **checksum**, which is computed when a client creates an HDFS file and stored in a separate hidden file in the HDFS namespace.

Note: implementations may differ

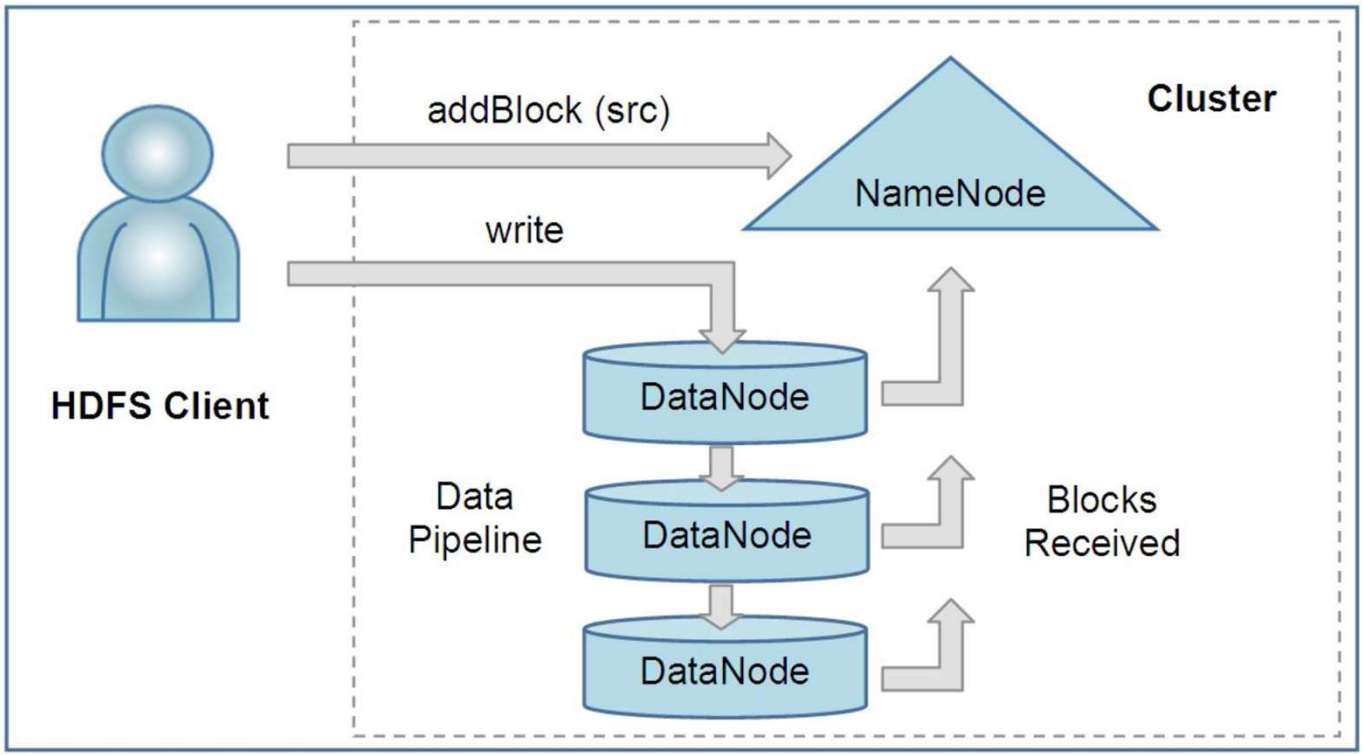
Robustness (2)

- FsImage and EditLog (journals) are central data structures of HDFS and thus the NameNode can be configured to support maintaining multiple copies of the FsImage and EditLog.
  - Synchronous updating of multiple copies of the FsImage and EditLog may degrade HDFS performance, but it is acceptable because HDFS applications are not metadata intensive (they are very data intensive in nature).
- The NameNode machine is a single point of failure for an HDFS cluster, but automatic restart and failover of the NameNode software to another machine can be implemented.
- Snapshots may be to roll back a corrupted HDFS instance to a previously known good point in time.

HDFS Data Organization

- Client retrieves a list of DataNodes on which to place replicas of a block
- Client writes block to the first DataNode
  - Client accumulates data locally until large enough to write a block
  - Client flushes the data block to the first DataNode in small portions (4KB)
- Writen in Pipeline:
  - The first DataNode forwards the data to the next DataNode in the list, which, in turn, writes that portion to its repository and then flushes that portion to the third DataNode, and so on
- When all replicas are written, the client moves on to write the next block in file

HDFS client read and write



An HDFS client creates a new file by giving its path to the NameNode. For each block of the file, the NameNode returns a list of DataNodes to host its replicas. The client then pipelines data to the chosen DataNodes, which eventually confirm the creation of the block replicas to the NameNode.

Source: The Hadoop Distributed File System, K. Shvachko, H. Kuang, S Radia, R. Chansler, IEEE MSST 2010

Hadoop in Real World

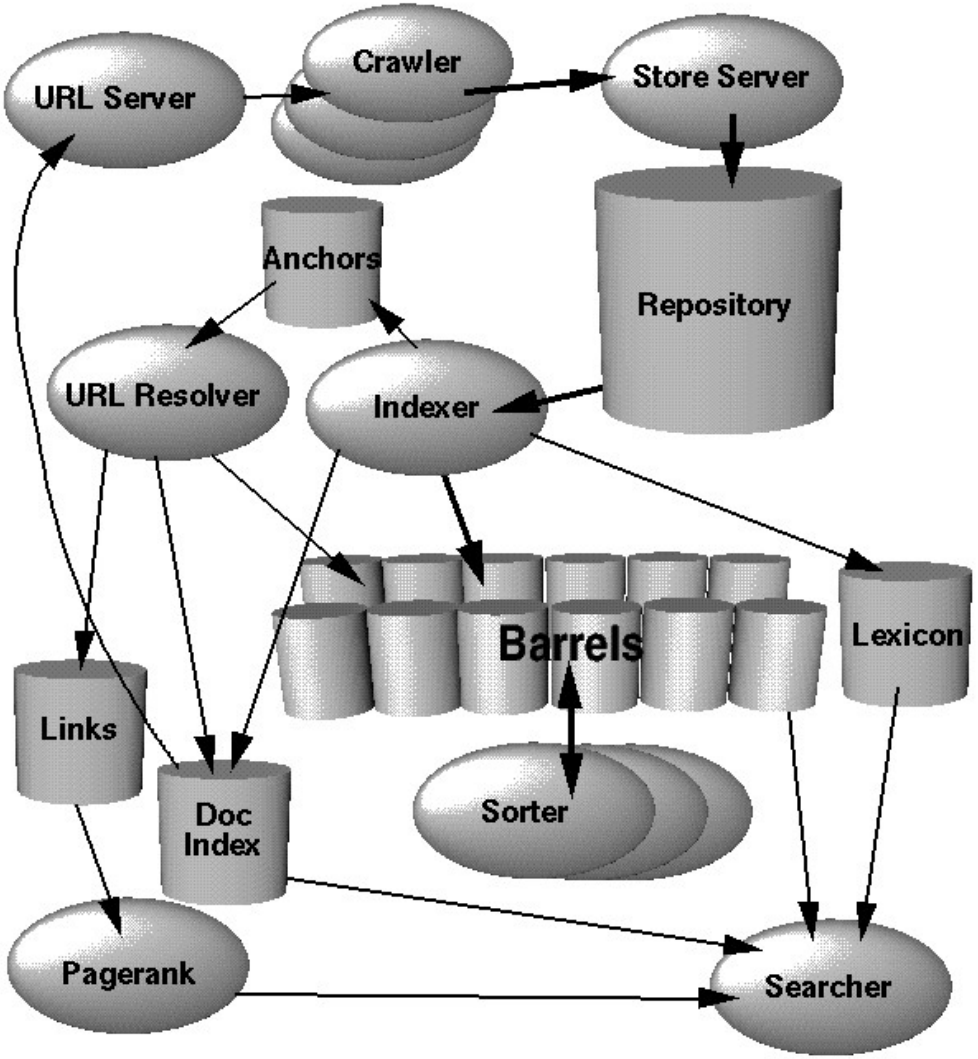
- Hadoop is in use at most organizations that handle big data:
  - Yahoo!
  - Facebook
  - Amazon
  - Netflix
  - Etc…
- Some examples of scale:
  - Hadoop clusters at Yahoo! span 25,000 servers, and store 25 petabytes of application data, with the largest cluster being 3500 servers (IEEEMSST 2010)
  - Yahoo!'s Search Webmap runs on 10,000 core Linux cluster and powers Yahoo! Web search
  - FB's Hadoop cluster hosts 100+ PB of data (July, 2012) & growing at ½ PB/day (Nov, 2012)

The Google File System

Reading: The Google File System, Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, ACM SOSP 2003

New file system: Colossus, 2013.

Google's Original Architecture



- Sorted barrels = inverted index
- Pagerank computed from link structure; combined with IR rank
- IR rank depends on TF, type of "hit", hit proximity, etc.
- Billion documents
- Hundred million queries a day

The Google File System

- Design Goal: a **scalable**, **fault-tolerant** **distributed** file system for large distributed data-intensive applications that runs on inexpensive commodity hardware and can deliver high aggregate performance to a large number of clients.
  - Largest clusters provided 300TB+ of storage across 1000+ nodes, heavily accessed by hundreds of clients (as of 2003).
- to be used mostly by co-designed applications, not by regular users

Design Assumptions

- System built from many inexpensive commodity components
  - Component failures are the norm
    - constant monitoring, error detection, fault tolerance, automatic recovery
- System will store a modest number of large files
    - e.g., repository of web documents
- Files mutated mostly by appending new data rather than overwriting existing data
  - **Once written, files are only read**
    - e.g., repository of web documents for future analysis
  - design focus will be on performance optimization and atomicity guarantees of appending, rather than random writes

Typical Usage

- a writer generates a file from beginning to end
  - when completion, atomically renames file to a permanent name, or
  - periodically checkpoints successfully written parts so that readers can use
- many writers concurrently append to a file for merged results or as a producer-consumer queue
  - **append-at-least-once** semantics preserves each writer's output
  - readers deal with the occasional padding and duplicates use the extra information (e.g., checksums, identifiers) provided by the writers

User interface

- Files organized in directories
- Usual primitives for
  - : *create, delete, open, close, read*, and *write*
- Two new operations
  - **Snapshots**
    - Create copies of files and directories
  - **Record appends**
    - Allow multiple clients to concurrently append data to the same file
    - Useful for implementing
      - Multi-way merge results
      - Producer-consumer queues
- not adhere to POSIX

POSIX (Portable Operating System Interface)

- POSIX defines the APIs, along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.
- A distributed file system is "**POSIX compliance**":
  - reads reflect any data previously written, and writes are atomic
    - multiple processes running on different nodes see the same behavior as if they were running on the same node using a local file system.
    - If the system has multiple buffer-caches, it needs to ensure cache consistency.
    - If data is striped across multiple data servers, reads and writes that span multiple stripes must be atomic.

GFS Architecture: Master-Slave

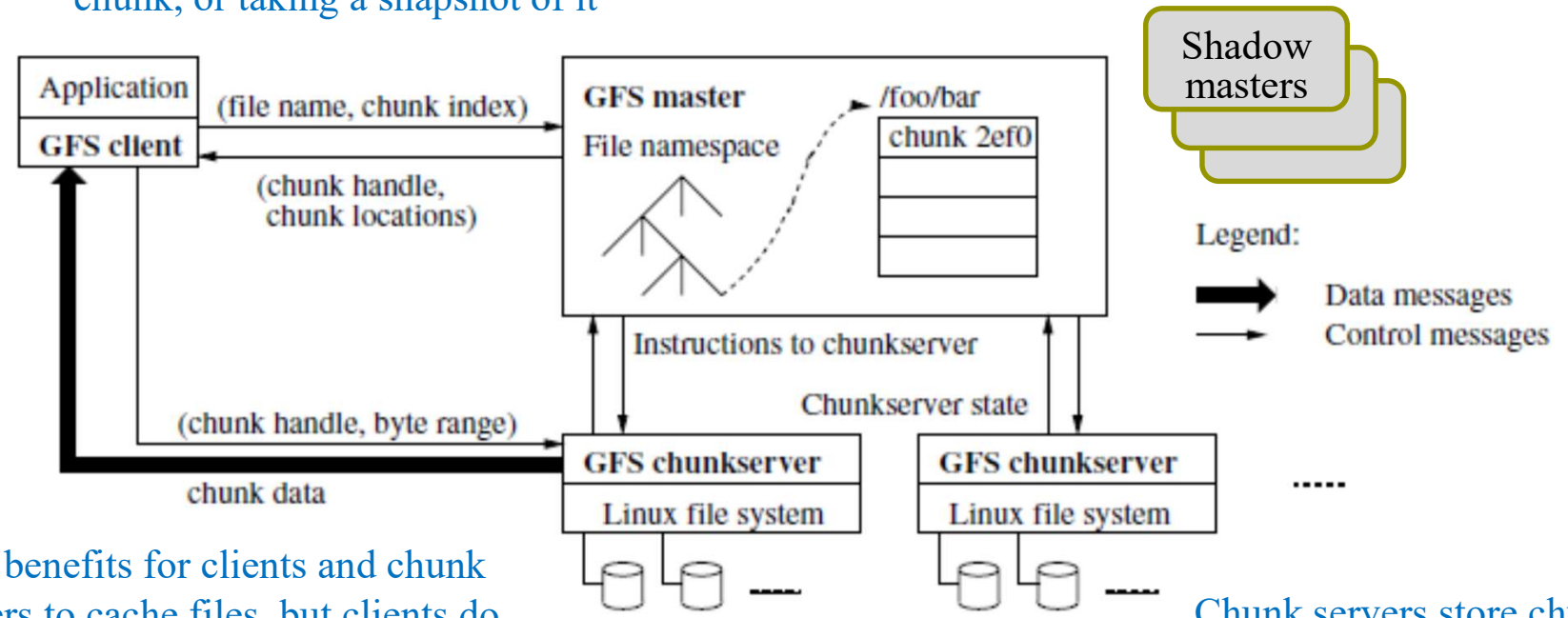
A file is divided into chunks of data

Master stores chunk-related metadata

- chunk (and their replicas) locations
- file names
- who are reading or writing to a particular chunk, or taking a snapshot of it

Master communicates with chunkservers through **heartbeats** and controls

- lease management
- garbage collection of orphaned chunks
- chunk migration between chunk servers



little benefits for clients and chunk servers to cache files, but clients do cache info about chunk handle and replica locations (note: local Linux file system already keeps frequently accessed data in memory)

Chunk servers store chunks locally as Linux files

source: The Google File System, S. Ghemawat, H. Gobioff, and S.-T. Leung, ACM SOSP 2003

Master Replication

- Master state is replicated for reliability.
  - Operation log and checkpoints are replicated on multiple machines.
  - Mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas.
- Use “Shadow” masters to enhance read
  - A shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does.
  - Like the primary, it polls chunkservers at startup (and infrequently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status.
  - So they may lag the primary slightly, typically fractions of a second, thus the name “shadow”, not “mirrors”.
  - They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results.

Files

- Files are divided into fixed-size ***chunks*** of 64MB
- Each chunk has a ***unique 64-bit id (chunk handle)***
  - assigned by the master at time of creation
  - stored by chunkservers on their local disks as Linux files
  - read/write by specifying chunk handle and byte range
  - Chunks are replicated (default: 3 replicas)
- all file system metadata are maintained by the master
  - namespace
  - access control information
  - mapping from files to chunks
  - locations of chunks

Chunk size

- Choose 64MB, much larger than typical file system block size
  - Pro.
    - Reduce the number of interactions between clients and master
      - suit for applications mostly read and write large files sequentially
    - reduce the number of TCP connection requests as clients are more likely to perform many operations on the same chunk
    - Reduce the size of the metadata stored on the master, thus allowing the metadata to be kept in memory
  - Con.(?)
    - Increase the likelihood of observing hot spots, esp. for small popular files
      - In practice not a problem and replication (increased if needed) helps

Metadata & The Master

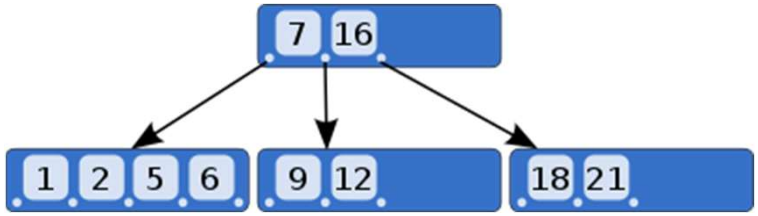
- Master stores in memory
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replica
- How much file storage size can be handled by a master of 4GB RAM?
  - each 64 MB chunk takes about 64 bytes of metadata and each file namespace data requires less then 64 bytes
- locations of chunk replicas are obtained from the chunkservers at master startup and whenever a chunkserver joins the cluster
  - Why not let master maintain a persistent view of chunk locations?

Operation Log

- Operation log: contains a historical record of critical metadata changes (*writes* or *record appends*) --- central to GFS.
- Files and chunks and their versions are all uniquely and eternally identified by the logical times at which they were created.
- File namespace mutations are atomic and handled exclusively by the master
  - the master does not acknowledge a client's change to operation log until it has replicated and kept the change persistent both locally and remotely.
  - checkpoint to reduce recover time
    - implemented in a compact B-tree

B+ Tree

B-tree



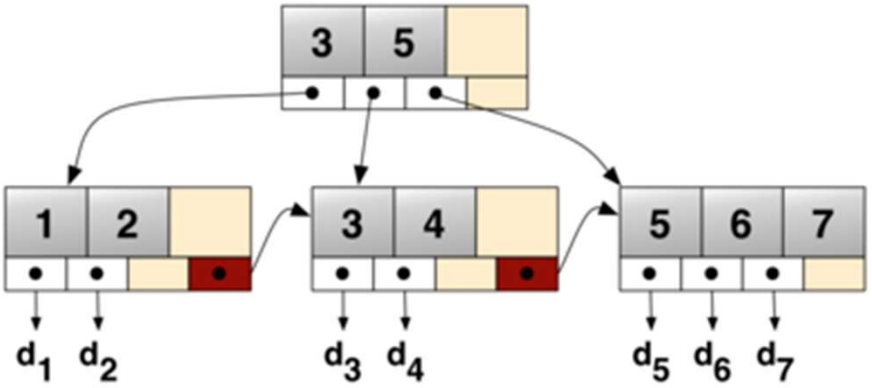
A B-tree of order m is a tree such that:

- Every node has at most m children.
- Every non-leaf node (except root) has at least $\lceil {}^{m}/_{2} \rceil$ child nodes.
- The root has at least two children if it is not a leaf node.
- A non-leaf node with k children contains $k - 1$ keys.
- All leaves appear in the same level

Source: wiki

A B-tree stores both keys and data in the internal and leaf nodes, while in a B+ tree internal nodes store only keys, and data are stored in the leaf nodes.

B+ -tree



Source: wiki

the leaf nodes are linked to provide ordered access to the data.

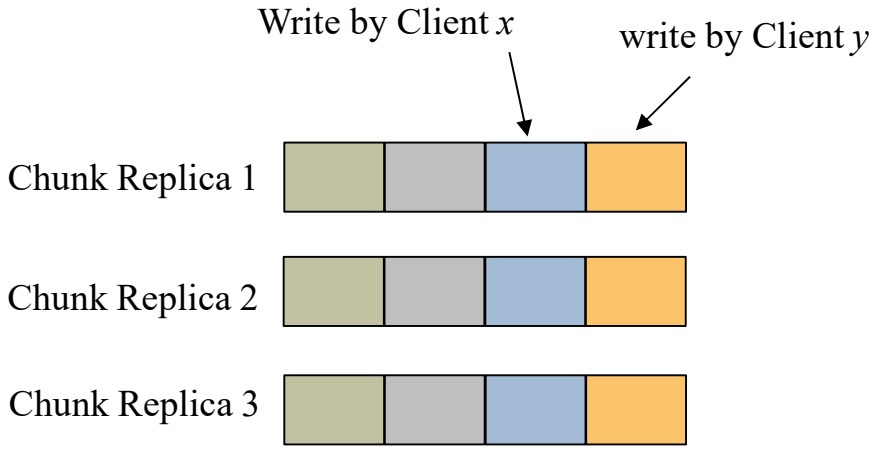
B+ trees are a most widely used index search data structure

Consistency model

- File namespace mutations are atomic and handled exclusively by the master
- Status of a file region after a data mutation:
  - **Consistent:** all clients will always see the same data regardless of which replica they read from
  - **Defined:** Consistent and clients will see what the mutation writes in its entirety
    - resulted from a successful mutation without interference from concurrent writers
  - **Undefined but Consistent**: all clients see the same data, but it may not reflect what any one mutation has written
    - resulted from concurrent successful mutations (but with mingled fragments)
  - **Inconsistent:** resulted from a failed mutation

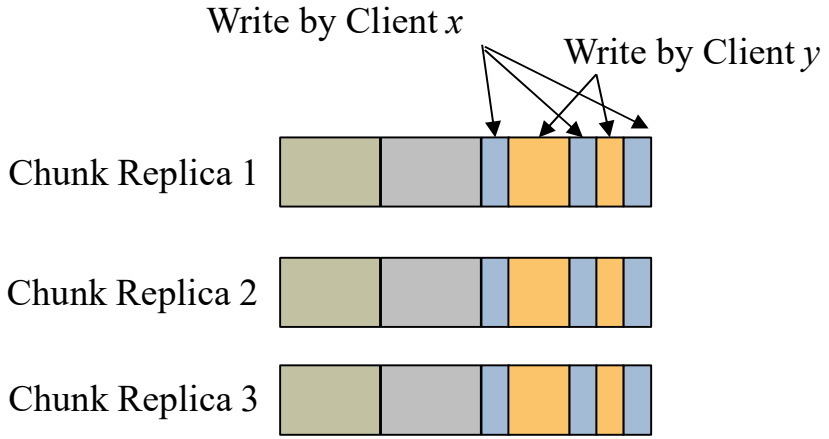
Defined vs. Undefined Regions

Defined



Undefined

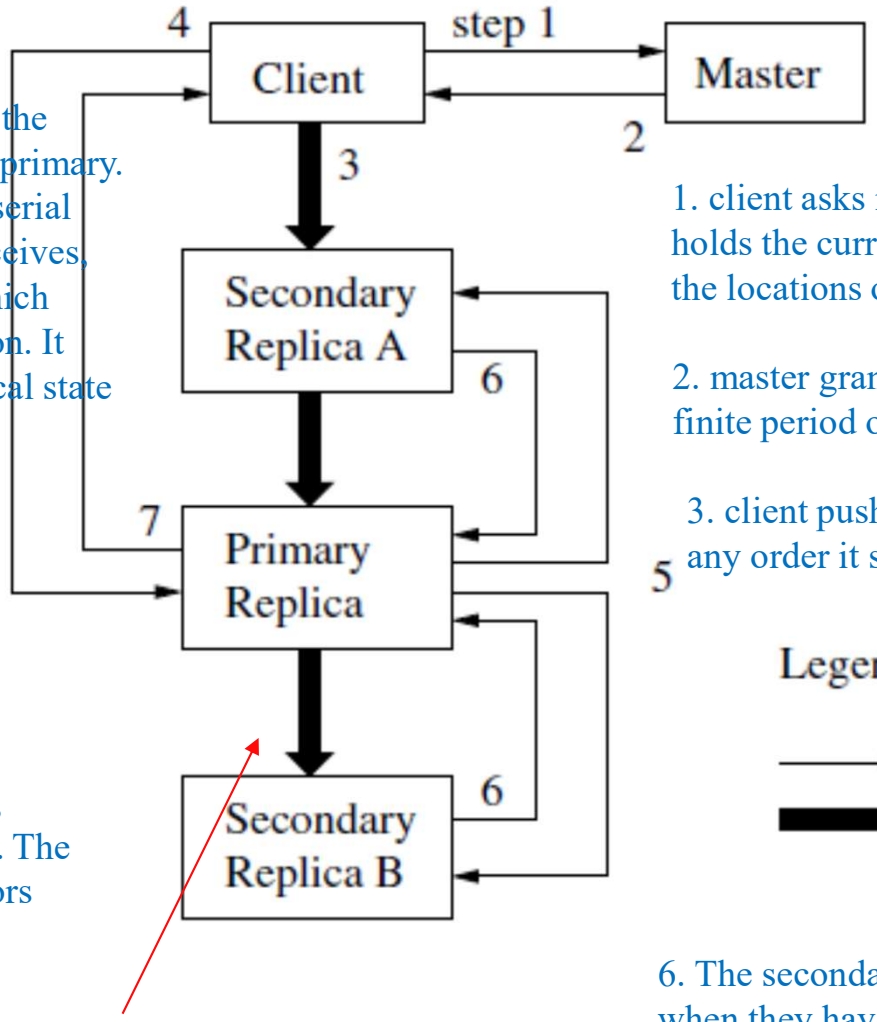
(due to successful concurrent writes)



Leases and Mutation Order

- All file namespace mutations are atomic
  - Handled exclusively by the master
- Use **leases** to maintain a consistent mutation order across replicas and minimize overhead at the master.
  - Master grants a chunk lease to one replica, call the **primary** (to assign a serial number to a mutation)
  - A lease has an initial timeout of 60 sec.
    - extension requests and grants are piggybacked on regular HeartBeat messages
    - master can safely grant a new lease to another replica after the old lease expires even it loses communication with a primary
- Global mutation order defined first by
  - lease grant order chosen by the master (inter-chunks)
  - serial numbers assigned by the primary within lease (intra-chunks)

Write Control and Data Flow



4. When all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.

5. The primary forwards the write request to all secondary replicas, each then applies mutations in the same serial number order assigned by the primary.

7. The primary replies to the client, including errors it has encountered. The client code needs to handle the errors (by retrying failed mutation).

1. client asks master which chunkserver holds the current lease for the chunk and the locations of the other replicas

2. master grants update permission for a finite period of time (60 seconds)

3. client pushes data to all the replicas, in any order it selects.

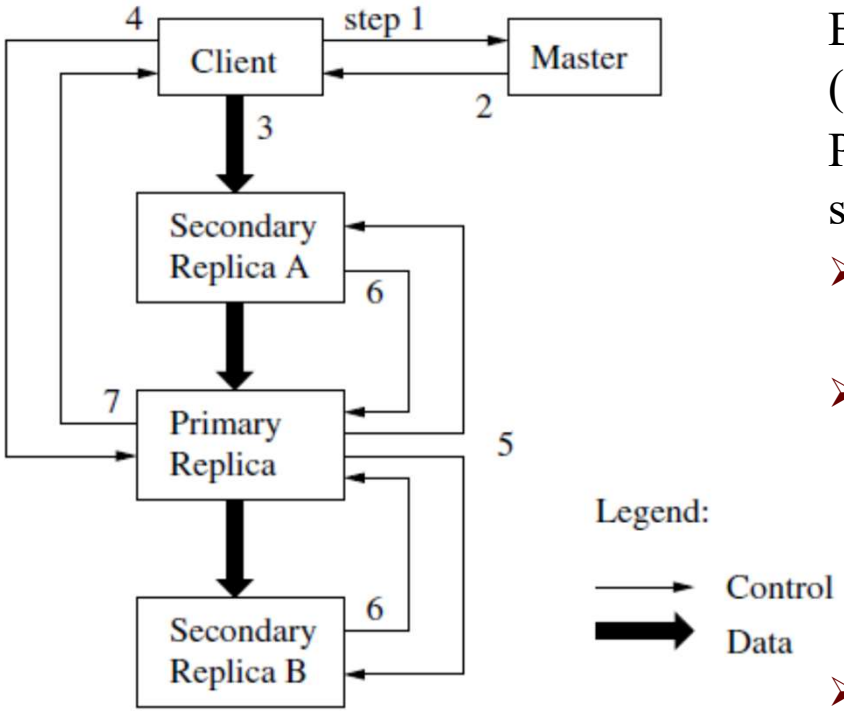
6. The secondaries acknowledge the primary when they have completed the operation.

Data is pushed linearly to chunkservers (each chooses the closest one to forward) in a pipelined fashion (start forwarding when some part are available) to fully utilize each machine's network bandwidth.

Atomic Record Appends

- Client specifies only the data to append, GFS will determine the offset (as GFS does not guarantee that all replicas are bytewise identical, thus replicas of a chunk may not end at the same position)
- GFS guarantees the semantics to be
  - At least once
  - Atomically
- Widely used to implement concurrent append to the same file
  - e.g., web crawling

Append Control and Data Flow

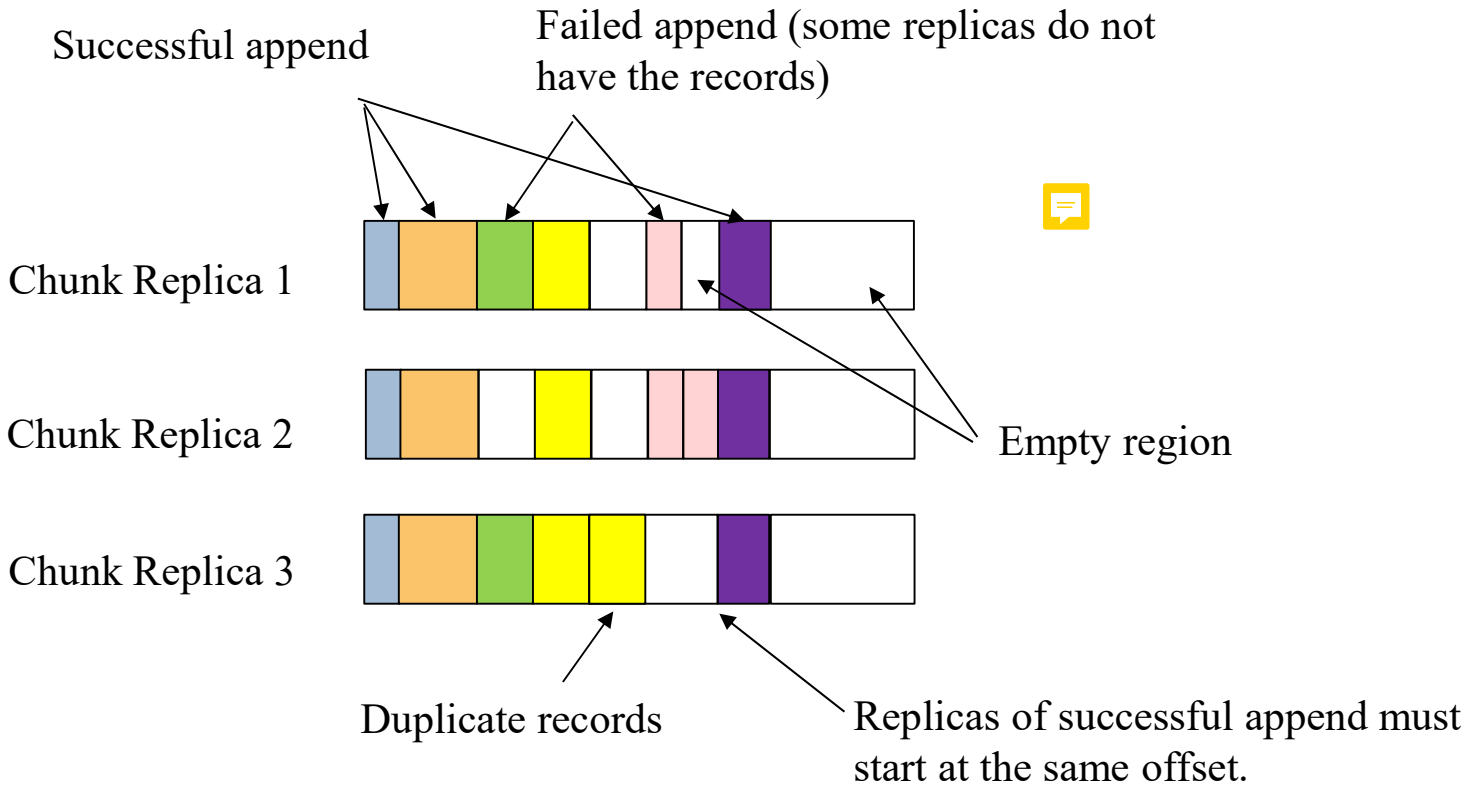


Basically the same as write operation, except in step (4) when the client sends its request to the primary:

Primary checks if record will exceed chunk max size, if so

- Pad chunk to maximum size, and tells secondaries to do the same
- Tell client to retry on a new chunk
  - Record append is restricted to be at most 1/4 of the maximum chunk size to keep worst case fragmentation at an acceptable level
- If a record append fails at any replica, the client retries the operation
  - so replicas of the same chunk may contain different data, e.g., duplicates of the same record in whole or in part
- any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary.

Possible Chunk Layout after Appends

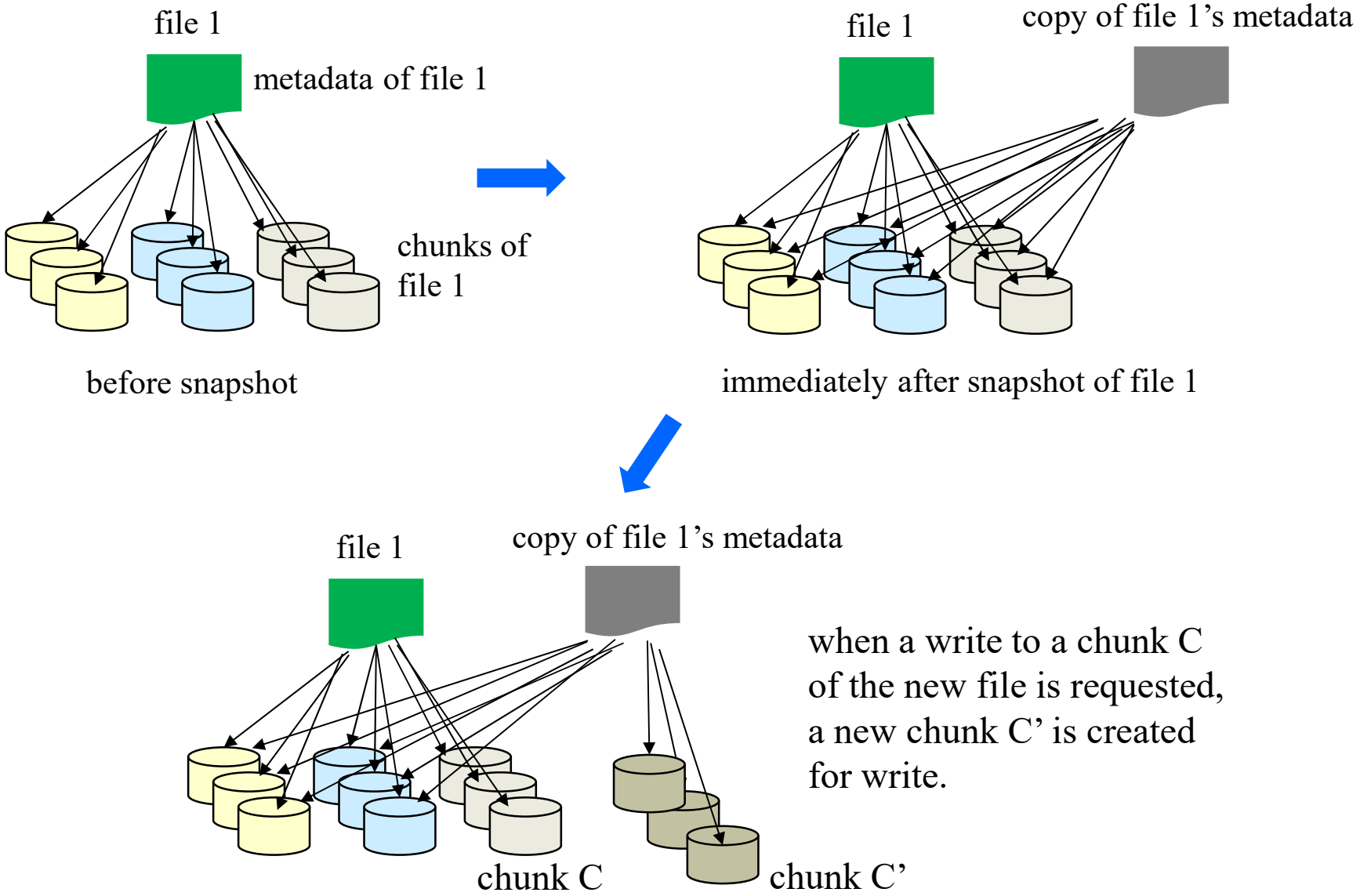


GFS does not guarantee that all replicas are bytewise identical

Snapshots

- Copy file and directories while minimizing any interruptions of ongoing mutation
  - How to make this operation almost instantaneously?
- Use ***copy-on-write*** approach
  - revokes any outstanding leases on the chunks in the files to snapshot
  - duplicating the metadata of the source
    - new copy currently share the same chunks as the source files
  - when master receives a write request to chunk C of the new copy
    - by noticing C's reference count greater than one
  - asks chunkservers holding a current replica of C to locally create a new chunk with a new handle

Snapshots Illustration

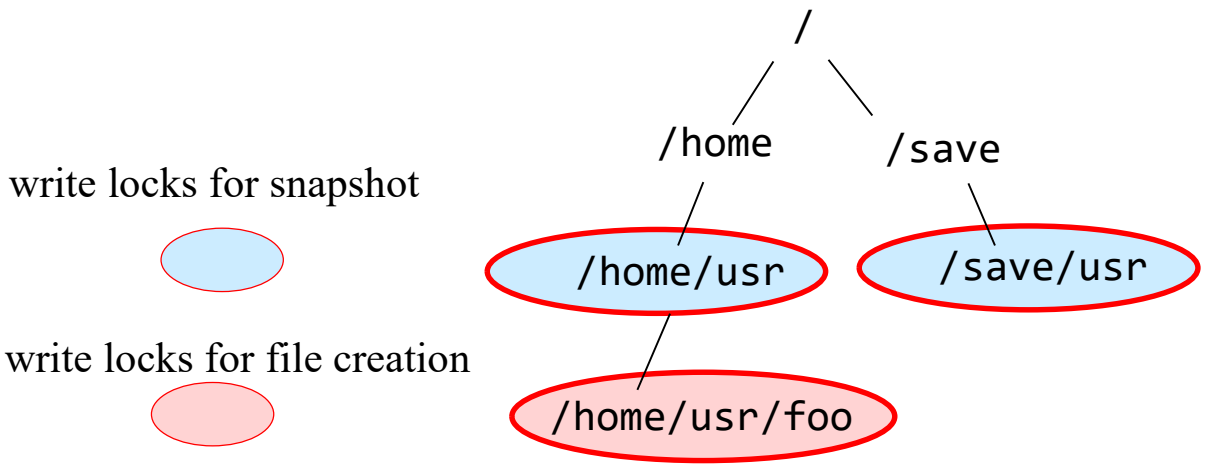


Namespace Management and Locking

- GFS logically represents its namespace as a lookup table mapping full pathnames to metadata.
  - /home/user/ ➔ …
  - /home/user/foo ➔ …
  - /home/user/bar ➔ …
  - prefix compression can reduce the table size to let it be efficiently represented in memory
- Each master operation acquires a set of locks
  - e.g., if it involves /d1/d2/.../dn/leaf, it will acquire read-locks on the directory names /d1, /d1/d2, ..., /d1/d2/.../dn, and either a read lock or a write lock on the full pathname /d1/d2/.../dn/leaf.
- Locks are acquired in a consistent total order to prevent deadlock
  - first ordered by level in the namespace tree, and
  - lexicographically within the same level

Namespace Management and Locking (2)

- Ex.: to create file /home/user/foo while /home/user is being snapshotted to /save/user
  - The snapshot operation acquires read locks on /home and /save, and write locks on /home/user and /save/user.
  - The file creation acquires read locks on /home and /home/user, and a write lock on /home/user/foo.
  - The two operations will be serialized properly because they try to obtain conflicting locks on /home/user.
  - file creation does not require a write lock on the parent directory because there is no "directory", or inode-like, data structure to be protected from modification
    - thus allowing concurrent mutations in the same directory



Replica Placement

- Chunk Replica Placement Policy
  - maximize data reliability and availability
  - maximize network bandwidth utilization
    - need to spread chunk replicas across racks

Chunk Creation, Re-replication, Rebalancing

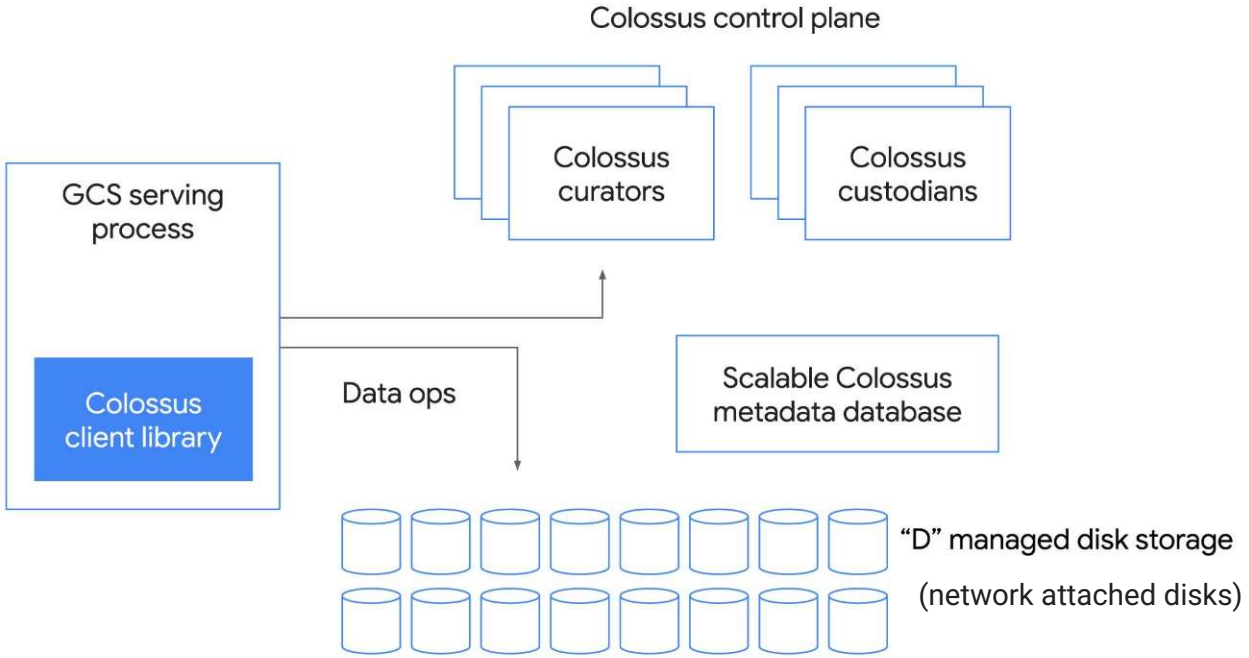
- Chunk Creation
  - Equalize disk utilization
    - place new replicas on chunkservers with below-average disk space utilization
  - Limit the number of “recent” creations on each chunkserver
  - Spread replicas across racks
- Re-replication
  - when the number of available replicas falls below a user-specified goal
- Rebalances replicas periodically
  - Move replica for better disk space and load balancing.

Garbage Collection

- For simplicity and reliability, when a file is deleted, GFS lazily reclaims the available physical storage during regular garbage collection at both the file and chunk levels
  - a deleted file is renamed to a hidden name that includes the deletion timestamp.
  - the master regularly scan the file system namespace
    - removes any such hidden files if they have existed for more than three days (interval configurable).
      - Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal.
  - the master also regularly scan the chunk namespace
    - erases the metadata of orphaned chunks (not reachable from any file)
    - each chunkserver reports the chunks it has through regular HeartBeat messages to the master,
      - the master replies with the identity of chunks that are no longer present in the master's metadata so that the chunkserver can safely delete them

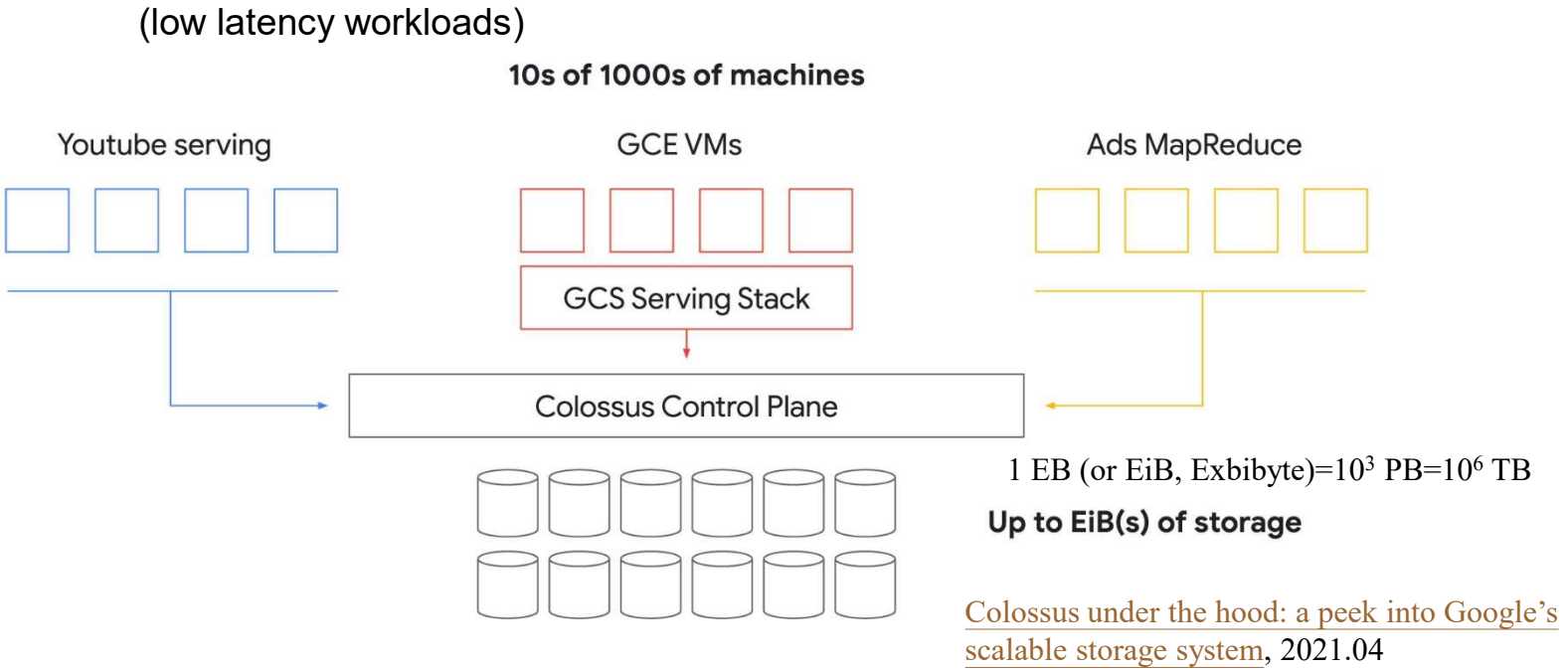
Google Colossus File System

- next-generation of the GFS
- enhances storage scalability and improves availability
- introduced a distributed metadata model
  - store metadata in BigTable
  - scale up by over 100x over the largest GFS clusters



Colossus under the hood: a peek into Google's scalable storage system, 2021.04

Typical Cluster



Colossus under the hood: a peek into Google's scalable storage system, 2021.04

- A single cluster is scalable to exabytes of storage and tens of thousands of machines
- The Colossus control plane provides the illusion that each application has its own isolated file system.
- Colossus uses a mix of flash and disk storage to meet any need of data access patterns and frequencies (e.g., hot data) and provides a range of service tiers.
- Applications use these different tiers by specifying I/O, availability, and durability requirements, and then provisioning resources (bytes and I/O) as abstract, undifferentiated units.