CPSC 436C
Cloud Computing for Data Science

Data store – Part1

Maryam R.Aliabadi

mraiyata@cs.ubc.ca

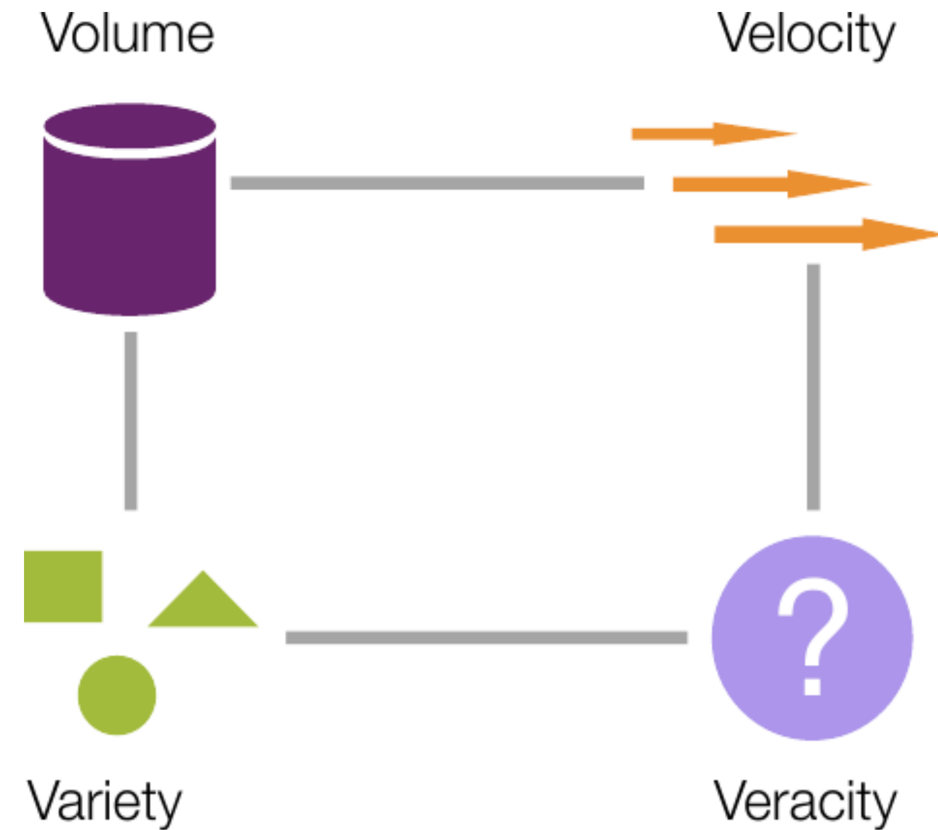
Spring 2024

Last Class‘ Review

- Big data
- Scale up Vs. Scale out
- Parallel architectures
- Big data analytics stack

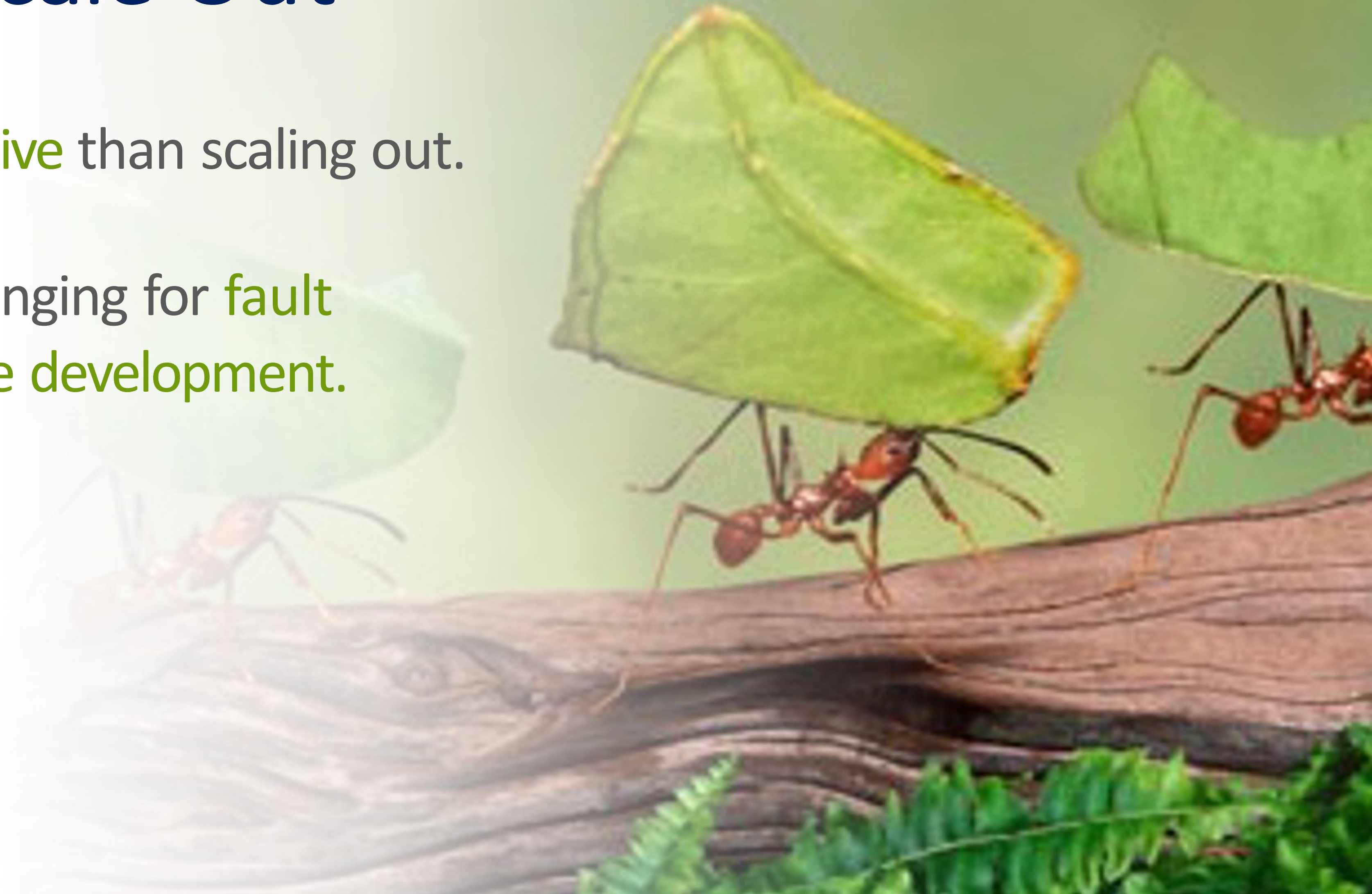
Big Data

- Volume: data size
- Velocity: data generation rate
- Variety: data heterogeneity
- Veracity: data quality



Scale Up vs. Scale Out

- Scale up: more expensive than scaling out.
- Scale out: more challenging for fault tolerance and software development.



Taxonomy of Parallel Architectures

Shared nothing

Shared disk

Shared memory

Interconnect

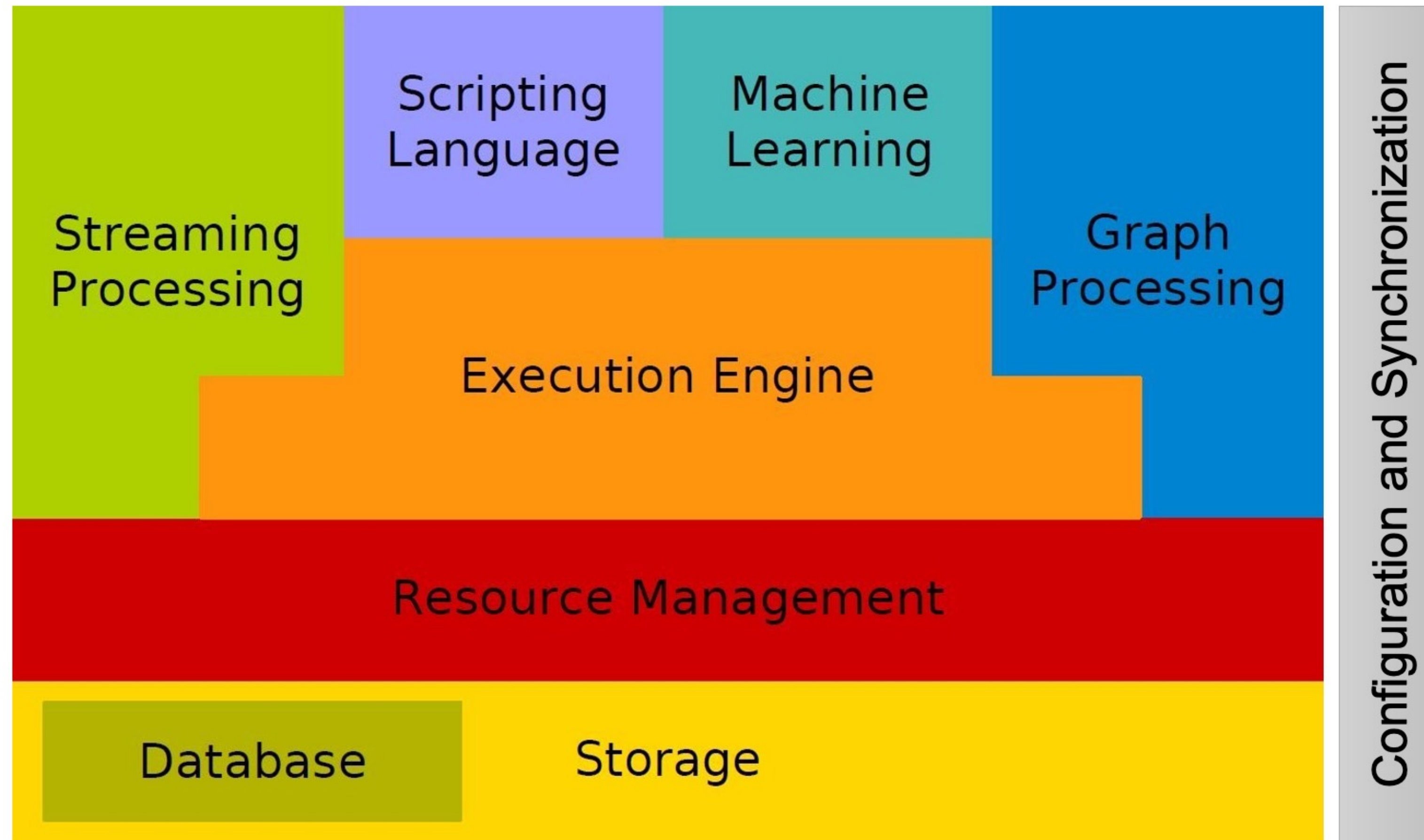
Interconnect

Interconnect

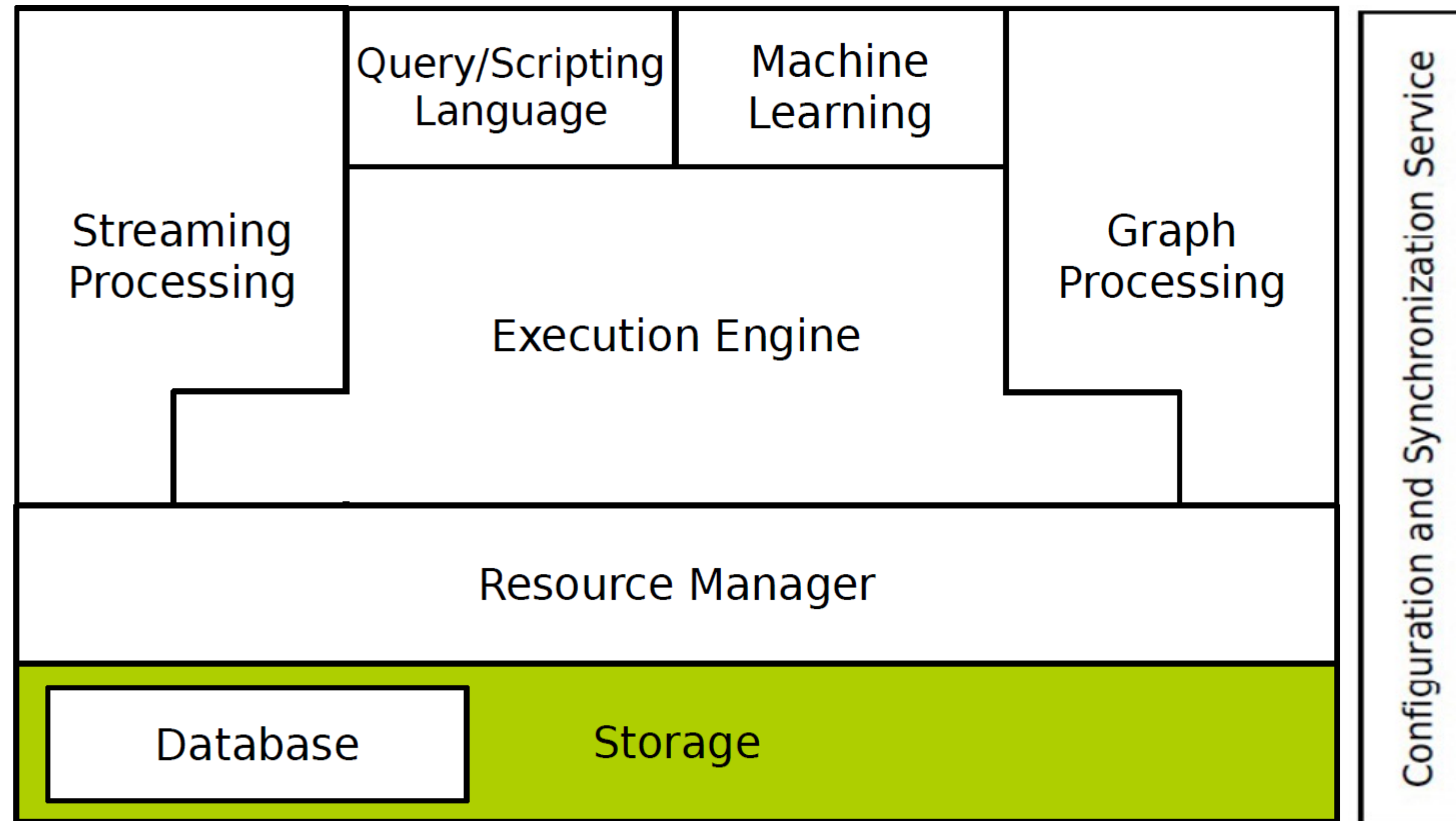
Process Memory Disk

DeWitt, D. and Gray, J. “Parallel database systems: the future of high performance database systems”. ACM Communications, 35(6), 85-98, 1992.

Big data analytics stack



Today's Topics



Data store

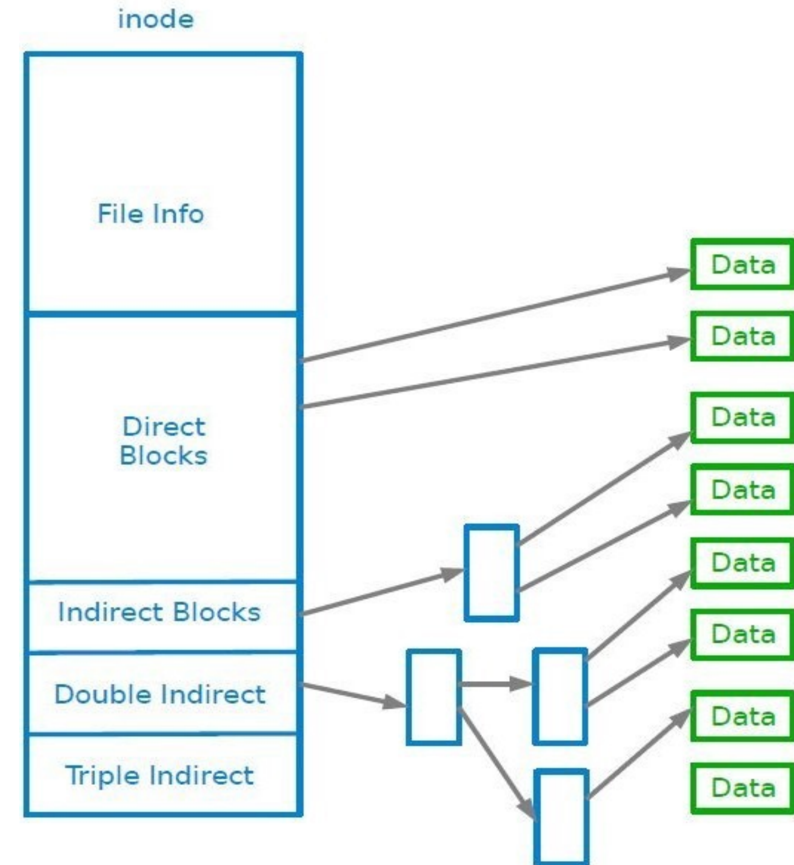
Data store

- Distributed file systems (DFS)
- SQL Databases
- NoSQL databases
- Key-Value store

Distributed File Systems

File System

- How to store and access files? File System
- What is a File System?
  - Controls how data is stored in and retrieved from disk.
  - Traditional: Unix file system



Distributed File Systems

- When data outgrows the storage capacity of a single machine.
- Partition data across a number of separate machines.
- Distributed file systems: manage the storage across a network of machines.

Benefits of DFS

- DFSs provide:

  1. File sharing over a network: without a DFS, we would have to exchange files by e-mail or use applications such as the Internet's FTP

  2. Transparent files accesses: A user's programs can access remote files as if they are local. The remote files have no special APIs; they are accessed just like local ones

  3. Easy file management: managing a DFS is easier than managing multiple local file systems

DSF Components

- DFS information can be typically categorized as follows:

    1. The data state: This is the contents of files
    2. The attribute state (meta data): This is the information about each file (e.g., file's size and access control list)
    3. The open file state:
       This includes which files are open or otherwise in use, as well as describing how files are locked

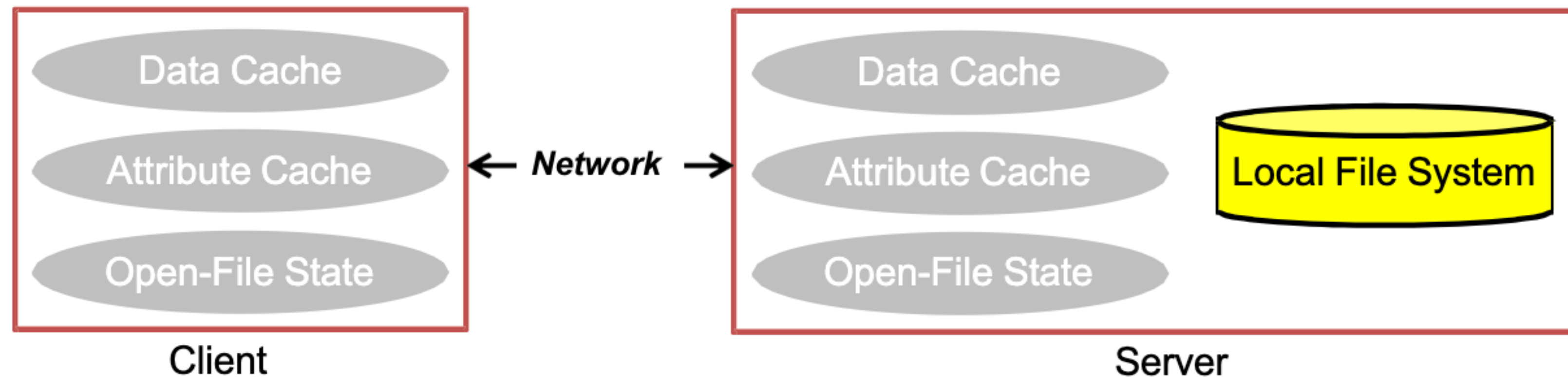
- Designing a DFS entails determining how its various components are placed. Specifically, by component placement we indicate:

    - What resides on the servers
    - What resides on the clients

DFS Component Placement

- The data and the attribute states permanently reside on the server's local file system, but recently accessed or modified information might reside in server and/or client caches.

- The open-file state is *transitory*; it changes as processes open and close files



DFS Examples

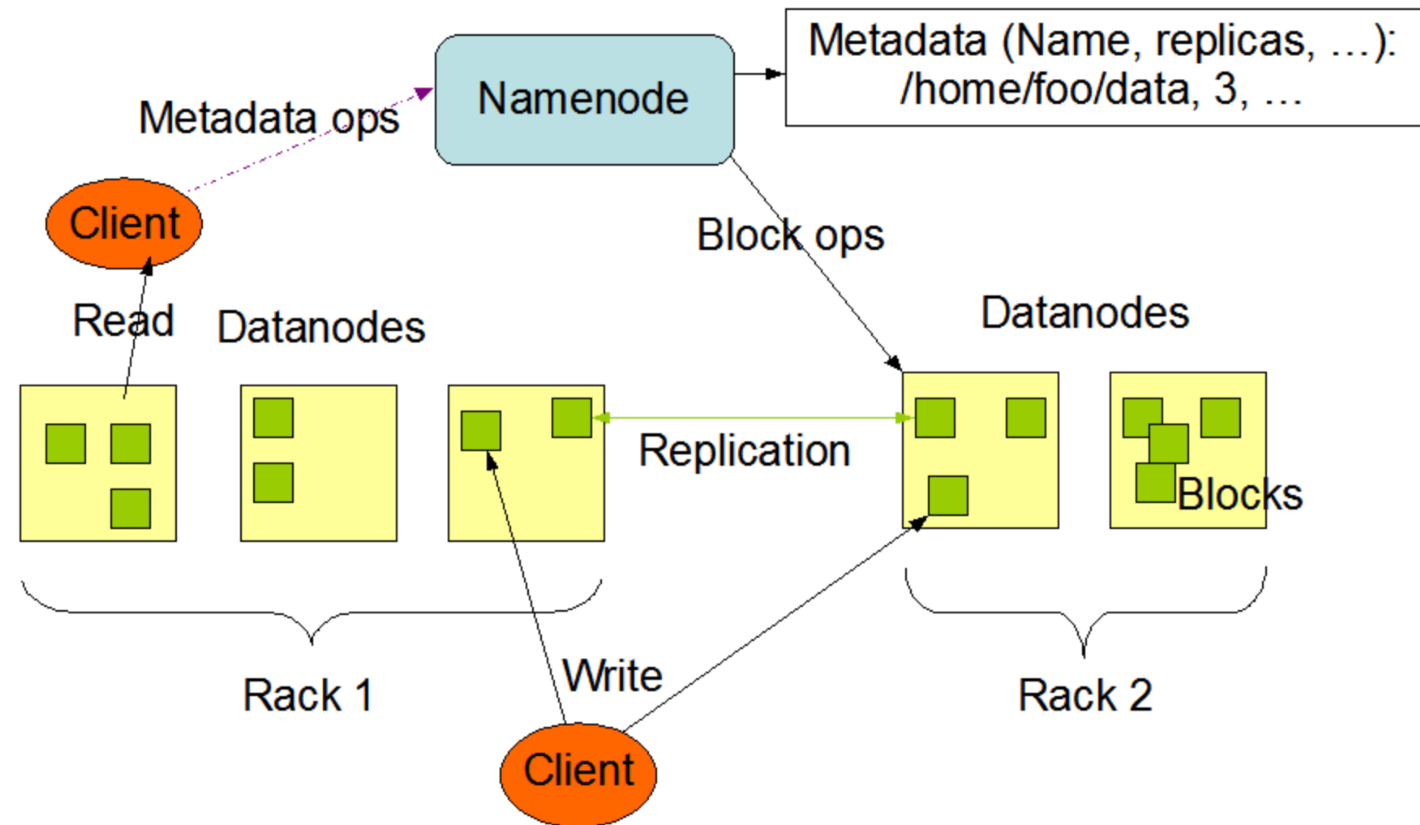
- Network File System (NFS)
- Storage Area Network (SAN)
- Google file System (GFS)
- Hadoop Distributed File System (HDFS)

HDFS

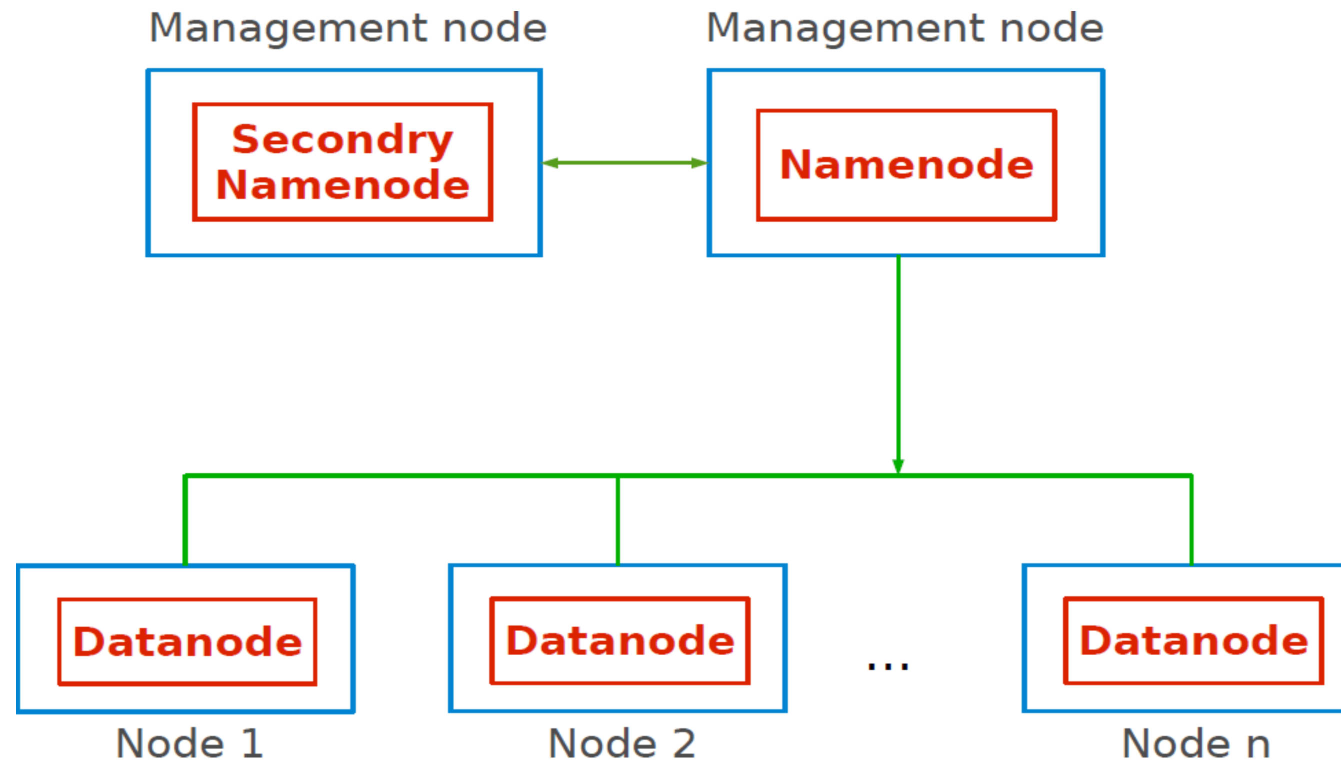
- Hadoop Distributed File System (HDFS)
- Appears as a single disk
- Runs on top of a native filesystem, e.g., ext3
- Fault tolerant: can handle disk crashes, machine crashes, ...
- Based on Google's filesystem GFS

HDFS Architecture

- Main components:
  - HDFS Name node
  - HDFS Data node
  - HDFS Client



HDFS Daemons

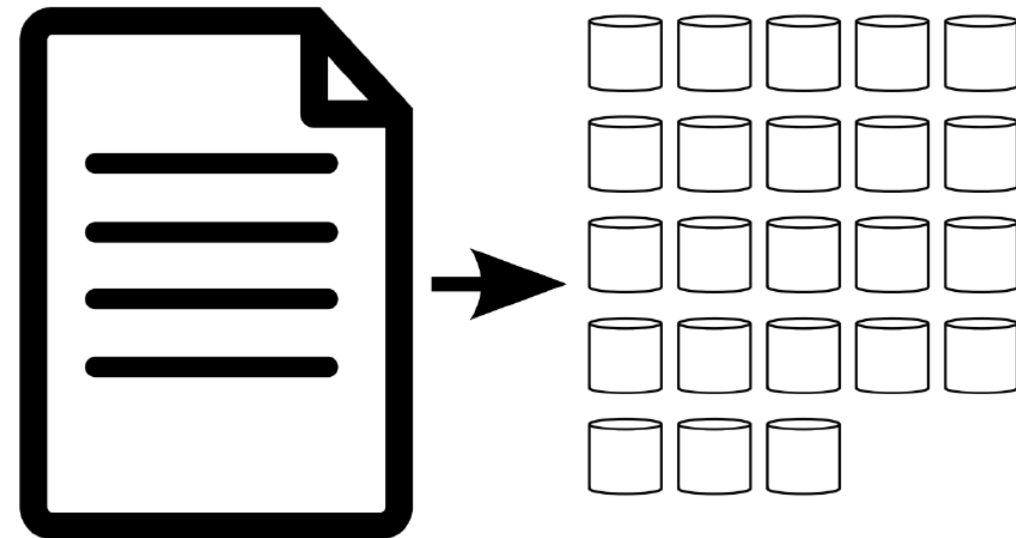


HDFS Daemons

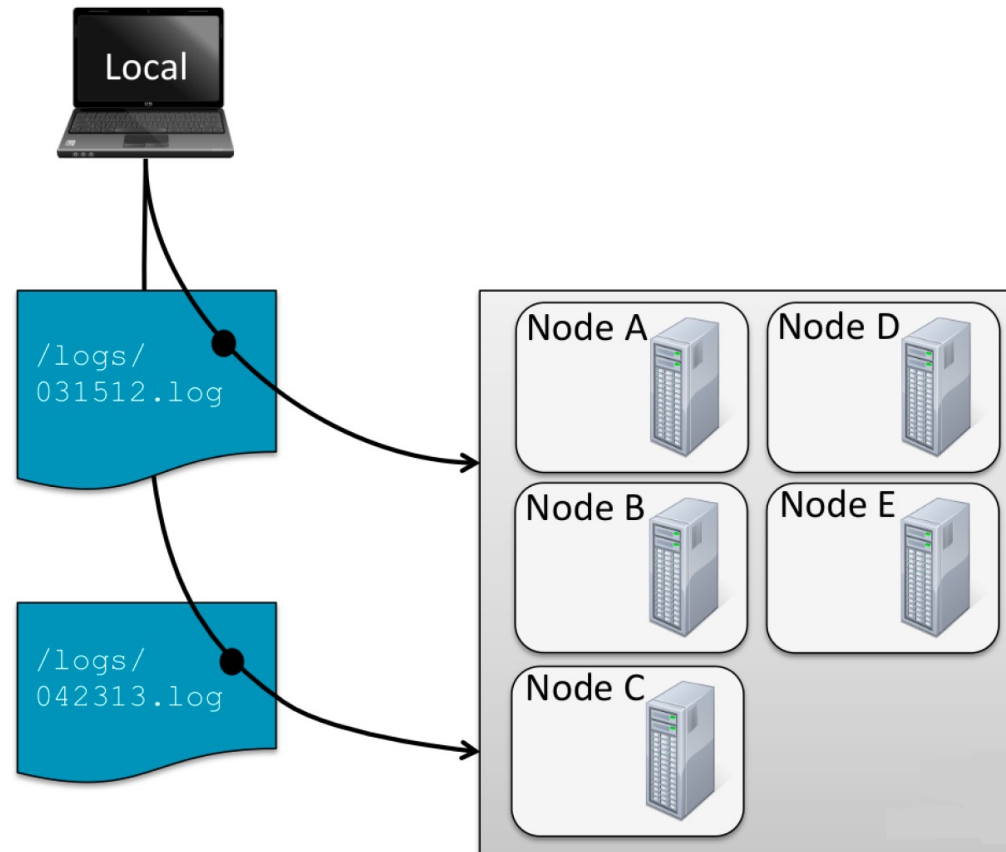
- HDFS cluster is manager by three types of daemons:
- Namenode
  - Manages the filesystem, e.g., namespace, meta-data, and file blocks
  - Metadata is stored in memory.
- Datanode
  - Stores and retrieves data blocks
  - Reports to Namenode
  - Runs on many machines
- Secondary Namenode
  - Only for checkpointing.
  - Not a backup for Namenode

Files and Blocks

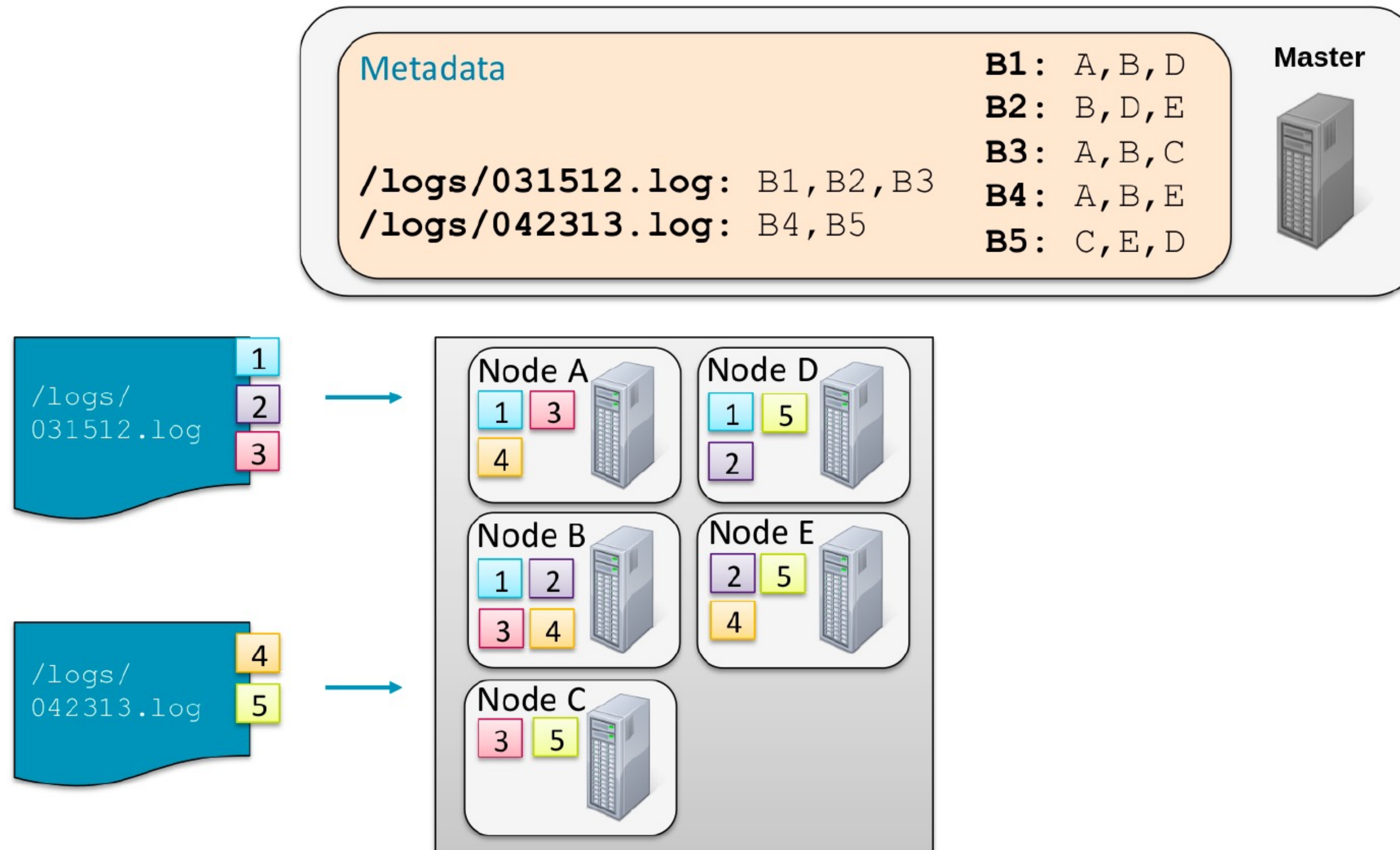
- Files are split into blocks.
- Blocks, the single unit of storage.
  - Transparent to user.
  - 64MB or 128MB.
- Same block is replicated on multiple machines: default is 3



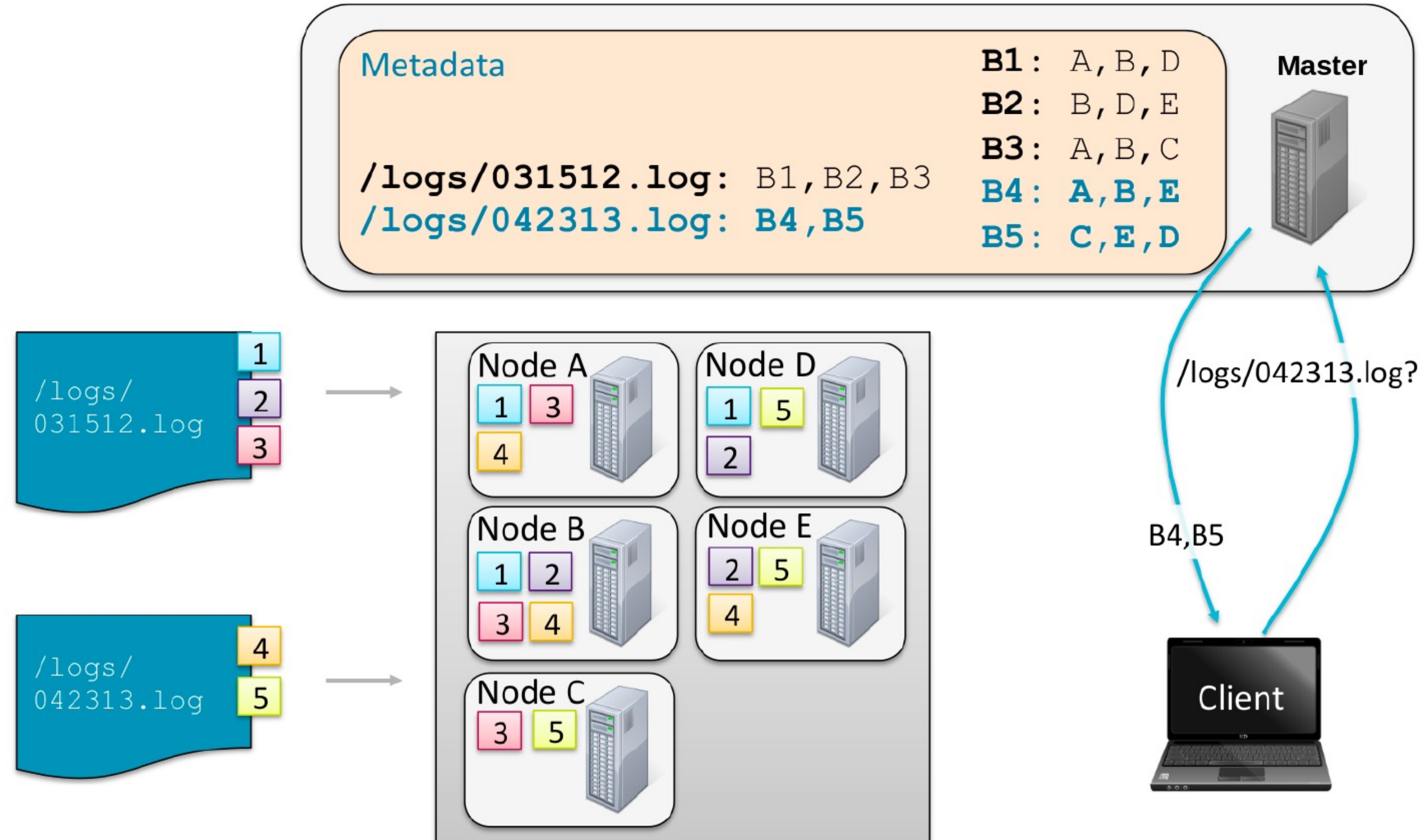
How to store and retrieve files?



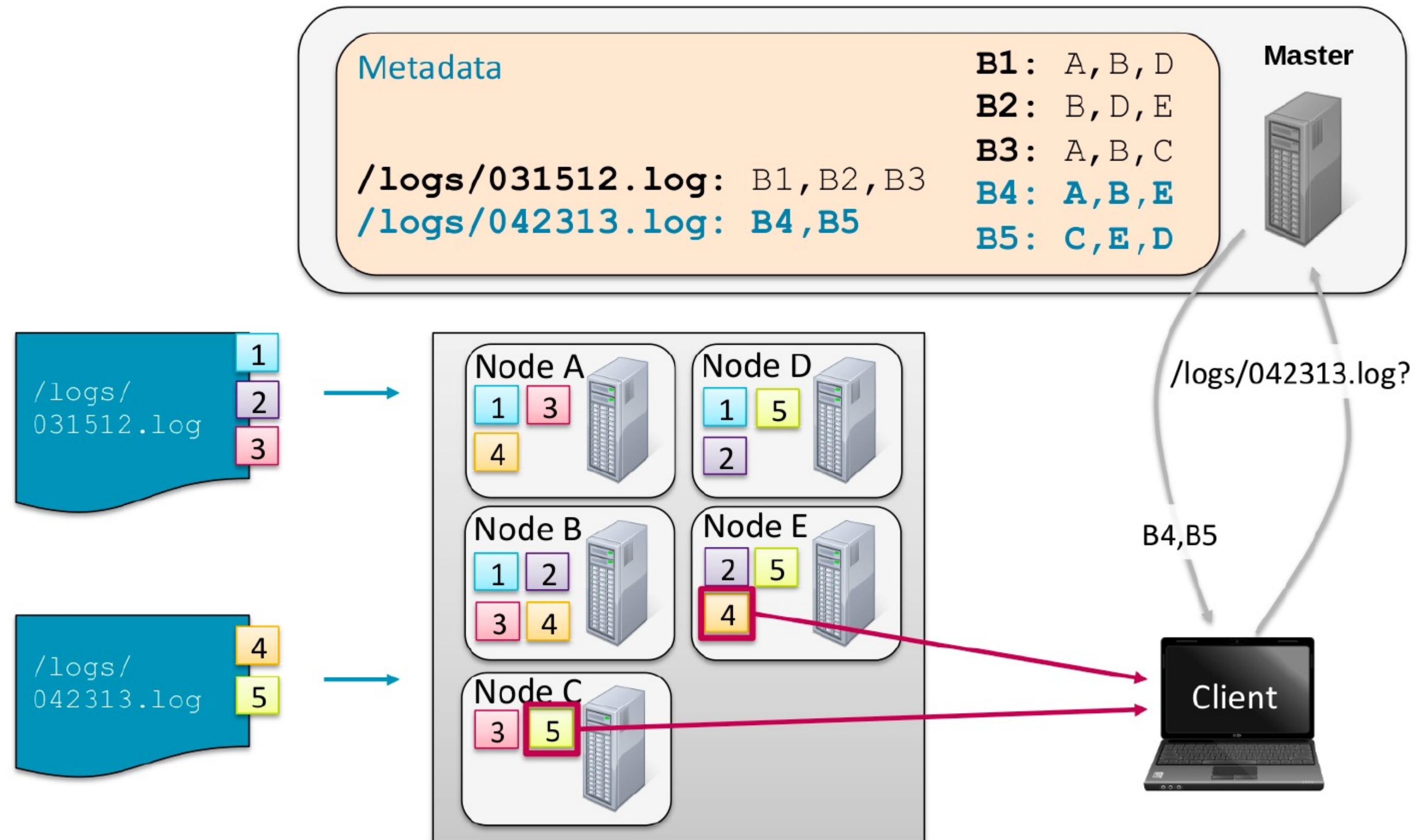
How to store and retrieve files?



How to store and retrieve files?

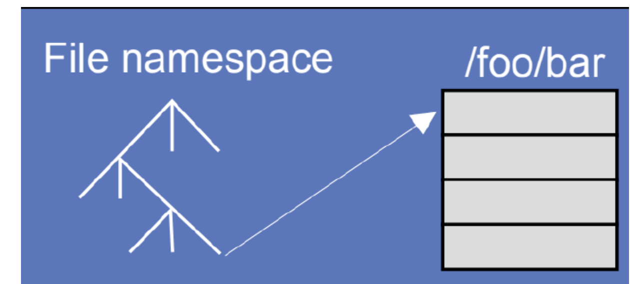


How to store and retrieve files?



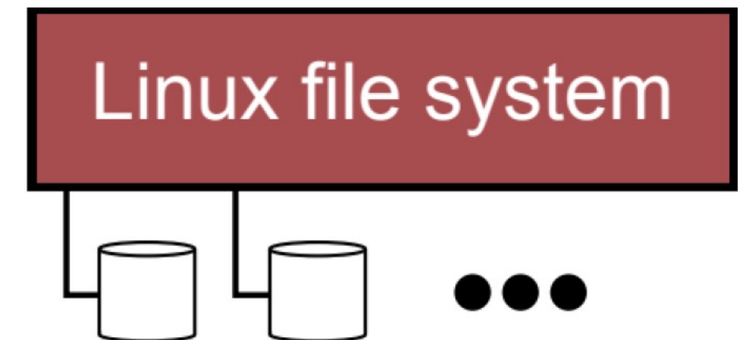
HDFS Namenode

- Responsible for all system-wide activities
- Maintains all file system metadata
  - Namespaces, ACLs (Access Control Lists), mappings from files to blocks, and current locations of blocks
  - All kept in memory, namespaces and file-to-block mappings are also stored persistently in operation log
- Periodically communicates with each Datanode
  - Determines block locations
  - Assesses state of the overall system



HDFS Datanode

- Manages blocks
- Tells Namenode what blocks it has
- Stores blocks as files
- Maintains data consistency of blocks



HDFS Client

- Client interacts with Namenode
  - To update the Namenode namespace.
  - To retrieve block locations for writing and reading.
- Client interacts directly with Datanode
  - To read and write data.
- Namenode does not directly write or read data.

Data Flow and Control Flow

- Data flow is decoupled from control flow
- Clients interact with the Namenode for metadata operations (control flow)
- Clients interact directly with Datanode for all files operations (data flow)

How big is a block?

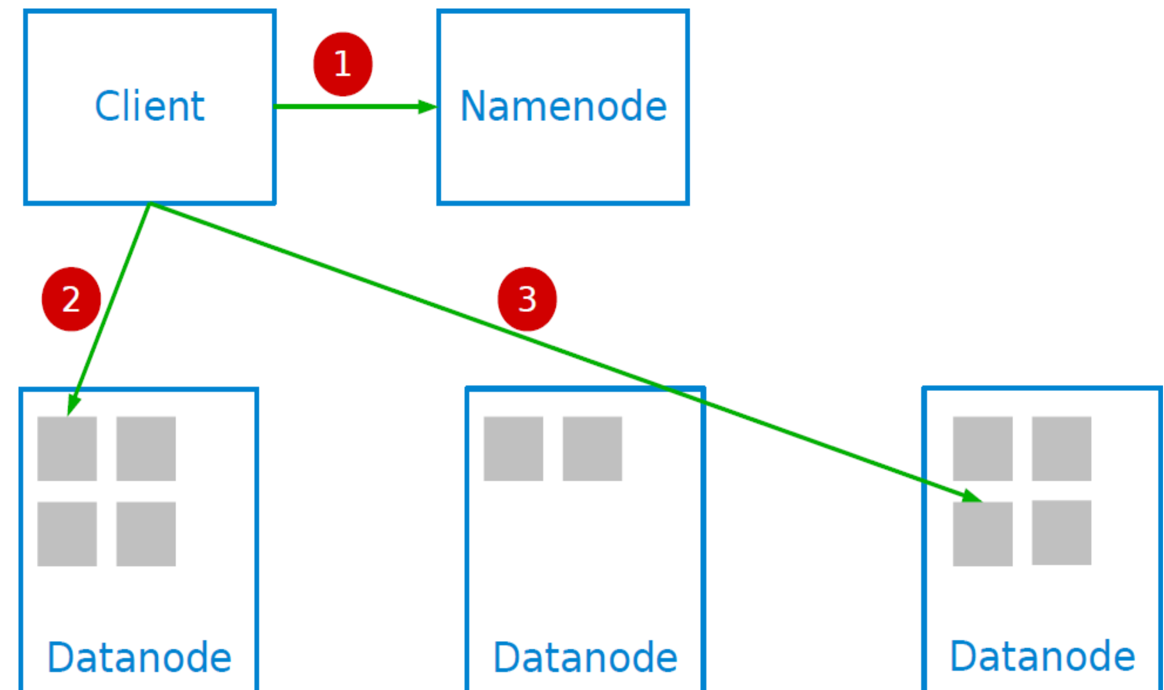
- 128MB (much larger than most file systems)

- *Advantages*
- Reduces the size of the metadata stored in Namenode
- Reduces clients' need to interact with Namenode

- *Disadvantages*
- Wasted space due to internal fragmentation

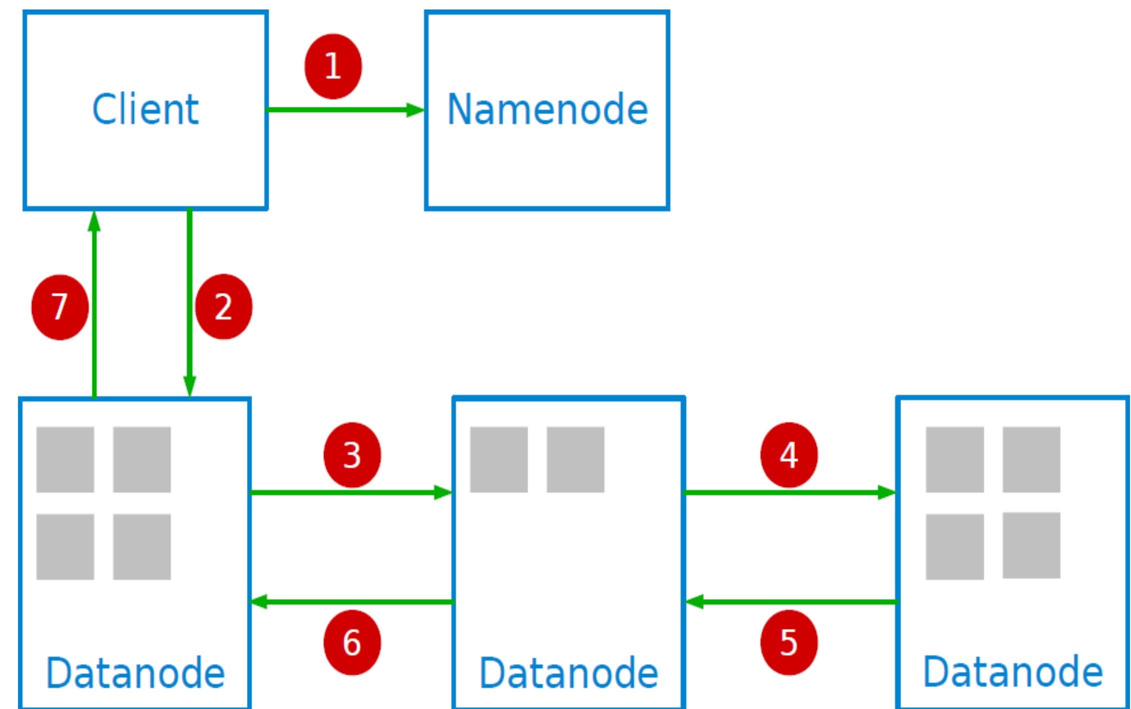
HDFS Read

- 1. Retrieve block locations.
- 2, 3. Read blocks to re-assemble the file.



HDFS Write

- 1. Create a new file in the Namenode's Namespace; calculate block topology.
- 2, 3, 4. Stream data to the first, second and third node.
- 5, 6, 7. Success/failure acknowledgment.



A Single Namenode

- The Namenode has a global knowledge of the whole system
  - It simplifies the design
  - The Namenode is (hopefully) never the bottleneck

- Clients never read and write file data through the Namenode
- Client only requests from Namenode which Datanodes to talk to
- Further reads of the same block do not involve the Namenode

The Namenode Operations

- Namespace management and locking
- Replica placement
- Creating, re-replicating and re-balancing replicas
- Garbage collection
- Stale replica detection

Namespace Management and Locking

- Represents its namespace as a lookup table mapping pathnames to metadata.
- Each master operation acquires a set of locks before it runs.
- Read lock on internal nodes, and read/write lock on the leaf.
- Example: creating multiple files (f1 and f2) in the same directory (/home/user/).
    - Each operation acquires a read lock on the directory name /home/user/
    - Each operation acquires a write lock on the file name f1 and f2
- Read lock on directory (e.g., /home/user/) prevents its deletion, renaming or snap-shot
- Allows concurrent mutations in the same directory

Replica Placement

- Based on locality and load balancing

- Maximize data reliability, availability and bandwidth utilization.

- Replicas spread across machines and racks, for example:
  - 1st replica on the local rack.
  - 2nd replica on the local rack but different machine.
  - 3rd replica on a different rack.

- The Namenode determines replica placement.

Creation, Replication and Balancing

- Creation
  - Place new replicas on Datanodes with below-average disk usage.
  - Limit number of recent creations on each Datanode.
- Re-replication
  - When number of available replicas falls below a user-specified goal.
- Rebalancing
  - Periodically, for better disk utilization and load balancing.
  - Distribution of replicas is analyzed.

Garbage Collection

- File deletion logged by Namnode.
- File renamed to a hidden name with deletion timestamp.
- Namenode regularly removes hidden files older than 3 days (configurable).
- Until then, hidden files can be read and undeleted.
- When a hidden file is removed, its in-memory metadata is erased.

Stale Replica Detection

- Block replicas may become stale: if a Datanode fails and misses mutations to the block while it is down.

- Need to distinguish between up-to-date and stale replicas.
- Block version number:
  - Increased when master grants new lease on the block.
  - Not increased if replica is unavailable.

- Stale replicas deleted by master in regular garbage collection.

Fault Tolerance

- Block replication (re-replication and re-balancing)

- Data integrity
  - Checksum for each block divided into 64KB chunks.
  - Checksum is checked every time an application reads the data.

Fault tolerance for Datanode

- All blocks are versioned.
- Version number updated when a new write is granted.
- Blocks with old versions are not served and are deleted.

Fault tolerance for Namenode

- Namenode state replicated for reliability on multiple machines.
- When master fails:
  - It can restart almost instantly.
  - A new Namenode process is started elsewhere.

- Shadow (not mirror) Namenode provides only read-only access to file system when primary Namenode is down.

HDFS is good for:

- Storing large files
  - Terabytes, Petabytes, etc...
  - 100MB or more per file.
- Streaming data access
  - Data is written once and read many times.
  - Optimized for batch reads rather than random reads.
- Cheap commodity hardware
  - No need for super-computers, use less reliable commodity hardware.

HDFS is not good for:

- Low-latency reads
  - High-throughput rather than low latency for small chunks of data.
  - HBase addresses this issue.
- Large amount of small files
  - Better for millions of large files instead of billions of small files.
- Multiple writers
  - Single writer per file.
  - Writes only at the end of file, no-support for arbitrary offset.

HDFS Vs. GFS

GFS	HDFS
Master	Namenode
Chunkserver	DataNode
Operation Log	Journal, Edit Log
Chunk	Block
Random file writes possible	Only append is possible
Multiple write/reader model	Single write/multiple reader model
Default chunk size: 64MB	Default chunk size: 128MB

Recap

- HDFS Architecture
- HDFS Components
- Name node operations
- HDFS API
- HDFS Fault tolerance

Next Topic:

Databases and Database Management Systems