# COMP9313: Big Data Management

Lecturer: Xin Cao

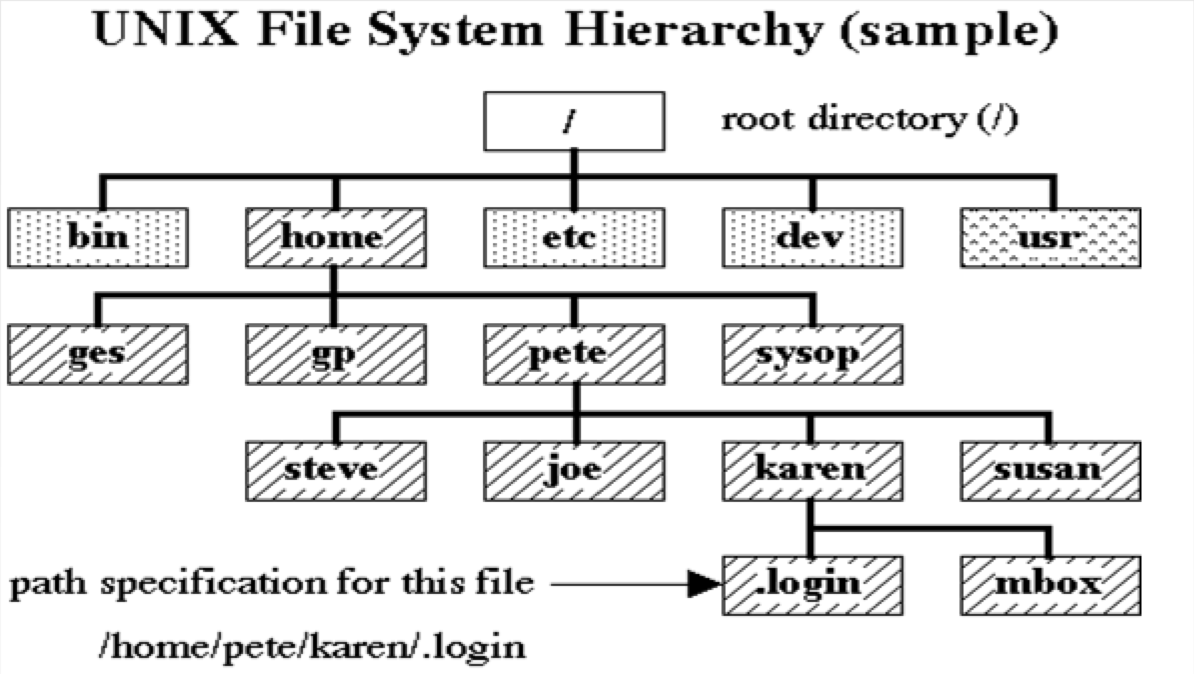
Course web site: http://www.cse.unsw.edu.au/~cs9313/

# Chapter 1.2 Introduction to HDFS, YARN, and MapReduce

# Part 1: HDFS

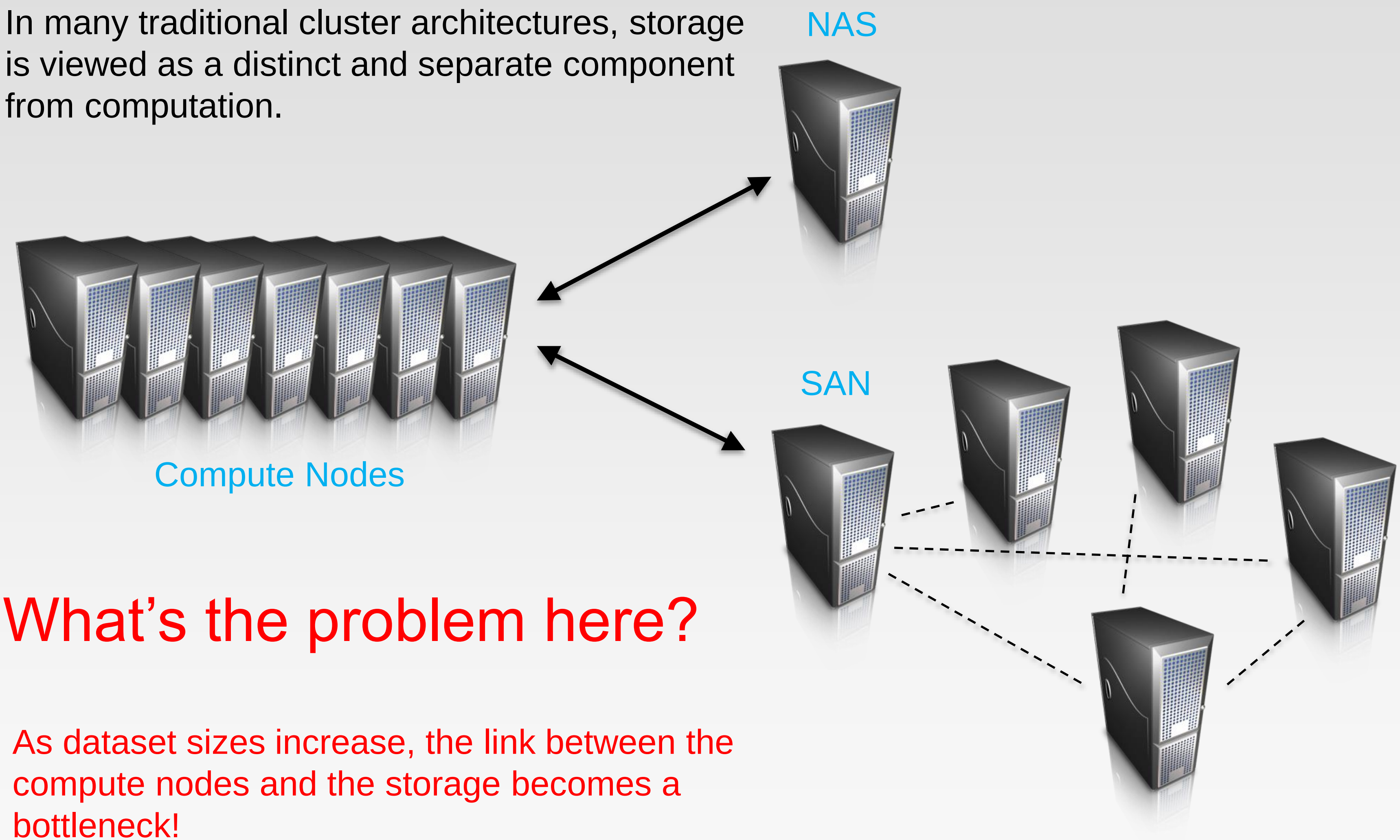
# File System

- A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk.



# How to Move Data to Workers?

In many traditional cluster architectures, storage is viewed as a distinct and separate component from computation.




What's the problem here?

As dataset sizes increase, the link between the compute nodes and the storage becomes a bottleneck!

# Latency and Throughput

- ***Latency*** is the time required to perform some action or to produce some result.
  - Measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.
  - I/O latency: the time that it takes to complete a single I/O.

- ***Throughput*** is the number of such actions executed or results produced per unit of time.
  - Measured in units of whatever is being produced (e.g., data) per unit of time.
  - Disk throughput: the maximum rate of sequential data transfer, measured by Mb/sec etc.

# Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - Not enough RAM to hold all the data in memory
  - Disk access is slow (low-latency), but disk throughput is reasonable (high throughput)
- A distributed file system is the answer
  - A distributed file system is a client/server-based application that allows clients to access and process data stored on the server as if it were on their own computer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

# Assumptions and Goals of HDFS

- Very large datasets
  - 10K nodes, 100 million files, 10PB
- Streaming data access
  - Designed more for batch processing rather than interactive use by users
  - The emphasis is on high throughput of data access rather than low latency of data access.
- Simple coherency model
  - Built around the idea that the most efficient data processing pattern is a write-once read-many-times pattern
  - A file once created, written, and closed need not be changed except for appends and truncates
- "Moving computation is cheaper than moving data"
  - Data locations exposed so that computations can move to where data resides

# Assumptions and Goals of HDFS (Cont')

- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Hardware failure is normal rather than exception. Detect failures and recover from them
- Portability across heterogeneous hardware and software platforms
  - designed to be easily portable from one platform to another

- HDFS is not suited for:
  - Low-latency data access (HBase is a better option)
  - Lots of small files (NameNodes hold metadata in memory)

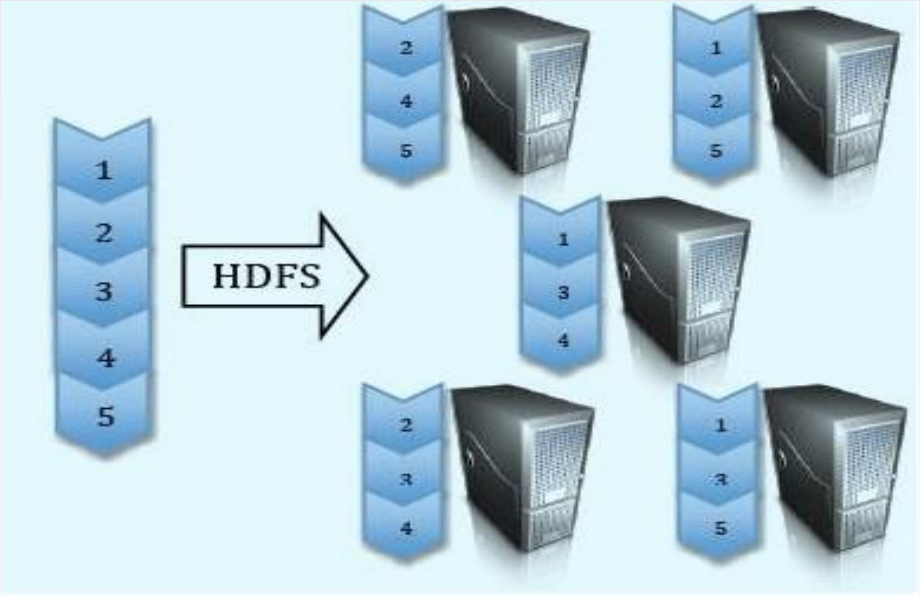
# HDFS Features

- The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.

- Basic Features:
  - Suitable for applications with large data sets
  - Streaming access to file system data
  - High throughput
  - Can be built out of commodity hardware
  - Highly fault-tolerant

# HDFS Architecture

- HDFS is a block-structured file system: Files broken into blocks of 64MB or 128MB
- A file can be made of several blocks, and they are stored across a cluster of one or more machines with data storage capacity.
- Each block of a file is replicated across a number of machines, To prevent loss of data.

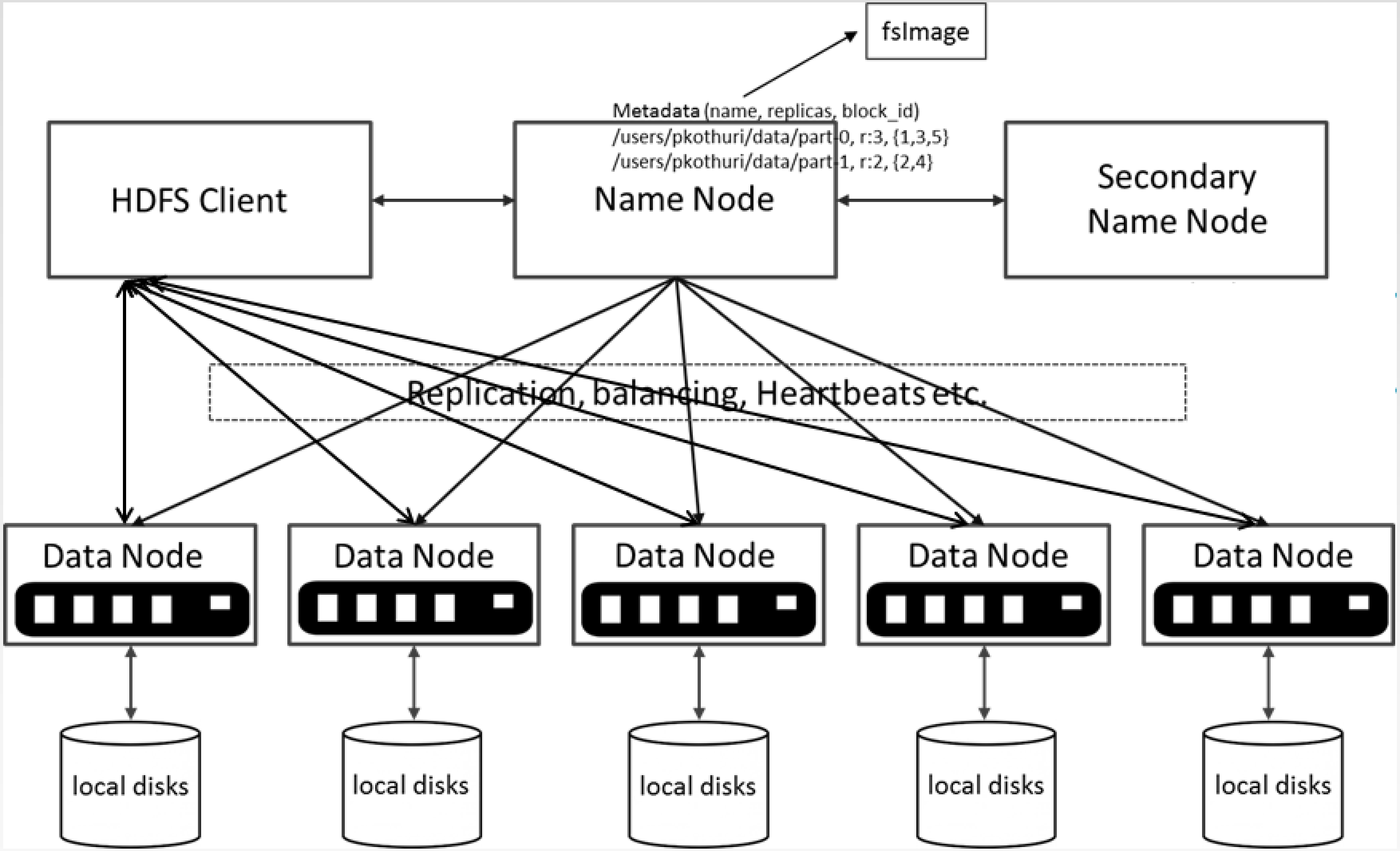


# HDFS Architecture

- HDFS has a master/slave architecture.

- There are two types (and a half) of machines in a HDFS cluster
  - NameNode: the heart of an HDFS filesystem, it maintains and manages the file system metadata. E.g., what blocks make up a file, and on which datanodes those blocks are stored.
    - Only one in an HDFS cluster
  - DataNode: where HDFS stores the actual data. Serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode
    - A number of DataNodes usually one per node in a cluster.
    - A file is split into one or more blocks and set of blocks are stored in DataNodes.
  - Secondary NameNode: **NOT** a backup of NameNode!!
    - Checkpoint node. Periodic merge of Transaction log
    - Help NameNode start up faster next time

# HDFS Architecture



# Functions of a NameNode

- Managing the file system namespace:
  - Maintain the namespace tree operations like opening, closing, and renaming files and directories.
  - Determine the mapping of file blocks to DataNodes (the physical location of file data).
  - Store file metadata.
- Coordinating file operations:
  - Directs clients to DataNodes for reads and writes
  - No data is moved through the NameNode
- Maintaining overall health:
  - Collect block reports and heartbeats from DataNodes
  - Block re-replication and rebalancing
  - Garbage collection

# NameNode Metadata

- HDFS keeps the entire namespace in RAM, allowing fast access to the metadata.
  - 4GB of local RAM is sufficient
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log (EditLog)
  - Records file creations, file deletions etc

# Functions 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.
- Periodically sends a report of all existing blocks to the NameNode (Blockreport)
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

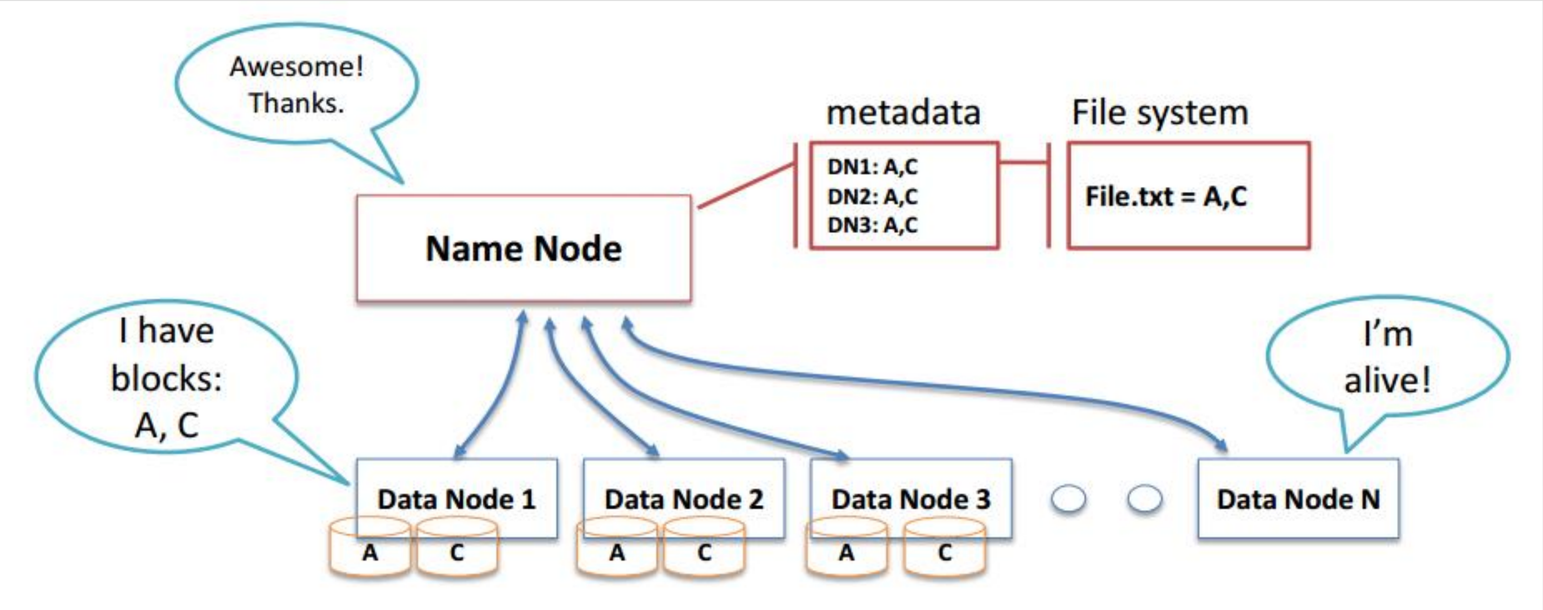
# Communication between NameNode and DataDode

- Heartbeats
  - DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available.
    - Once every 3 seconds
  - The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them

- Blockreports
  - A Blockreport contains a list of all blocks on a DataNode

- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster periodically

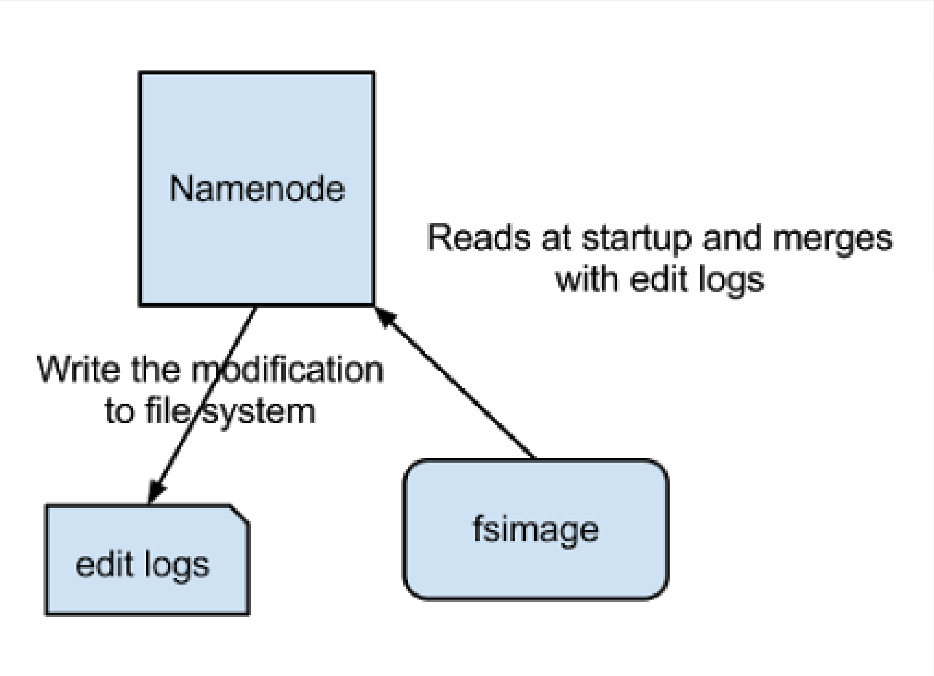
# Communication between NameNode and DataDode




- TCP – every 3 seconds a Heartbeat
- Every 10$^{th}$ heartbeat is a Blockreport
- Name Node builds metadata from Blockreports
- If Name Node is down, HDFS is down

# Inside NameNode

- FsImage - the snapshot of the filesystem when NameNode started
  - A master copy of the metadata for the file system
- EditLogs - the sequence of changes made to the filesystem after NameNode started

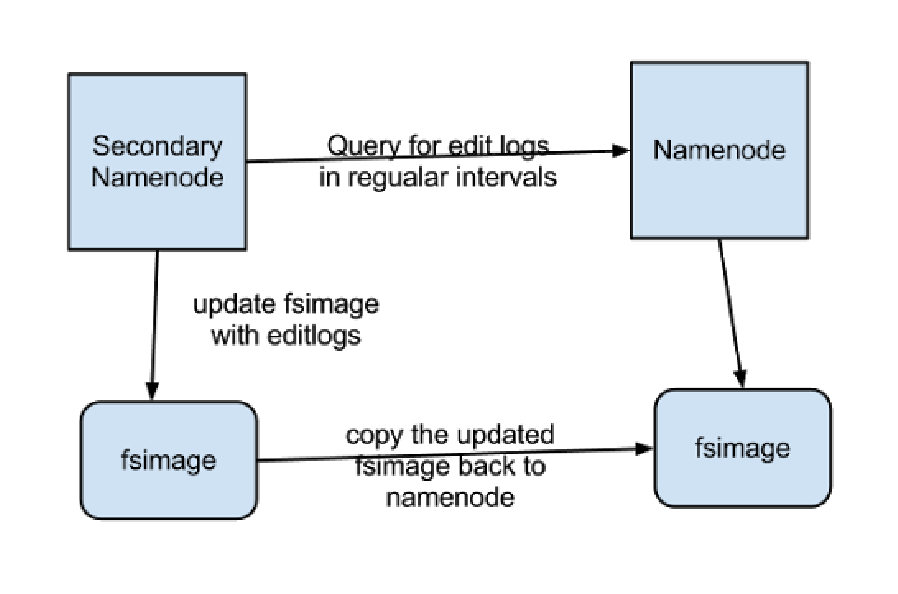


# Inside NameNode

- Only in the restart of NameNode, EditLogs are applied to FsImage to get the latest snapshot of the file system.
- But NameNode restart are rare in production clusters which means EditLogs can grow very large for the clusters where NameNode runs for a long period of time.
  - EditLog become very large , which will be challenging to manage it
  - NameNode restart takes long time because lot of changes has to be merged
  - In the case of crash, we will lose huge amount of metadata since FsImage is very old
- How to overcome this issue?

# Secondary NameNode

- Secondary NameNode helps to overcome the above issues by taking over responsibility of merging EditLogs with FsImage from the NameNode.
  - It gets the EditLogs from the NameNode periodically and applies to FsImage
  - Once it has new FsImage, it copies back to NameNode
  - NameNode will use this FsImage for the next restart, which will reduce the startup time



# File System Namespace

- Hierarchical file system with directories and files
  - /user/comp9313
- Create, remove, move, rename etc.
- NameNode maintains the file system
- Any meta information changes to the file system recorded by the NameNode (EditLog).
- An application can specify the number of replicas of the file needed: replication factor of the file.

# HDFS Commands

- All HDFS commands are invoked by the bin/hdfs script. Running the hdfs script without any arguments prints the description for all commands.

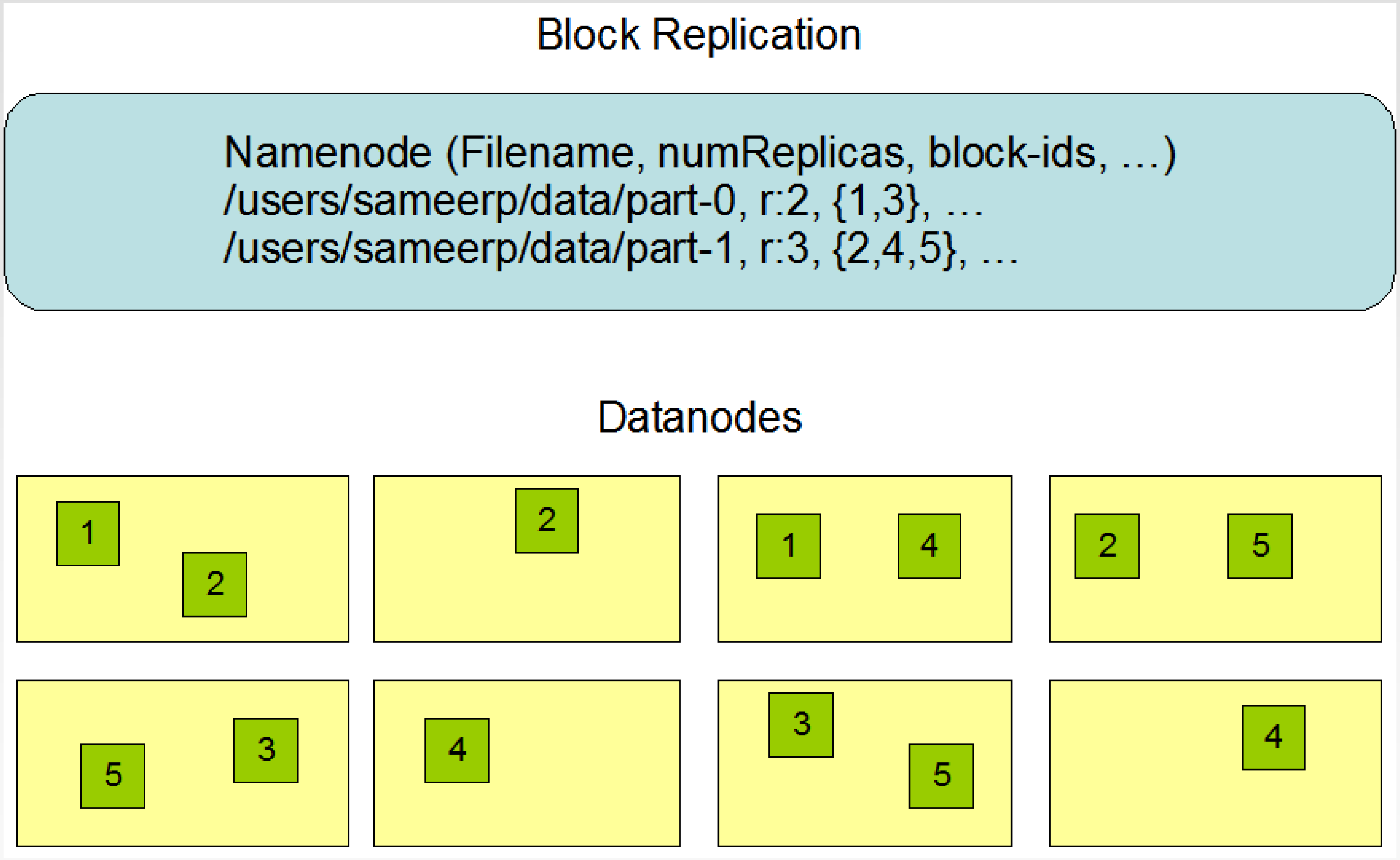
- Usage: hdfs [SHELL_OPTIONS] COMMAND [GENERIC_OPTIONS] [COMMAND_OPTIONS]

  - hdfs dfs [COMMAND [COMMAND_OPTIONS]]

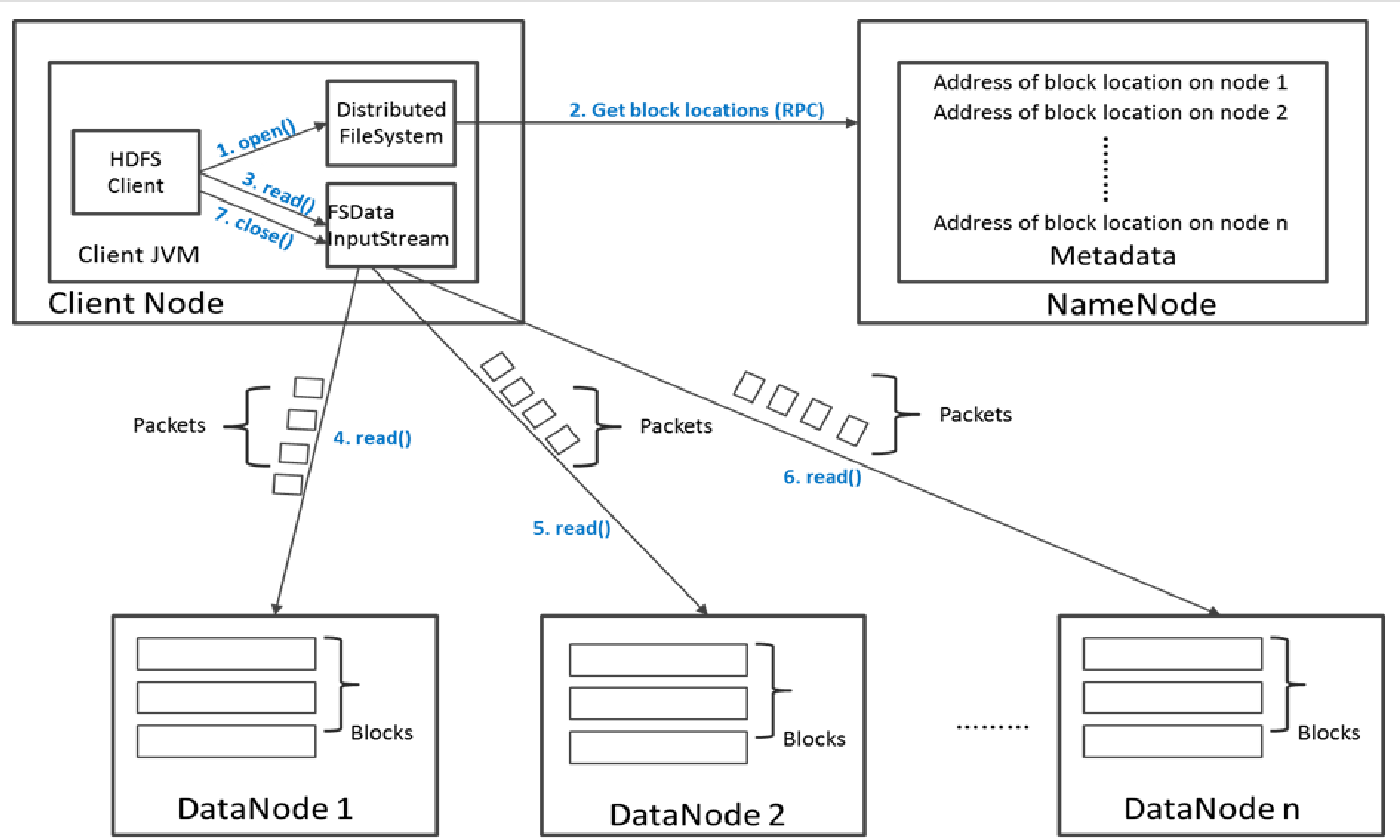
  - Run a filesystem command on the file system supported in Hadoop. The various COMMAND_OPTIONS can be found at File System Shell Guide.

# Data Replication

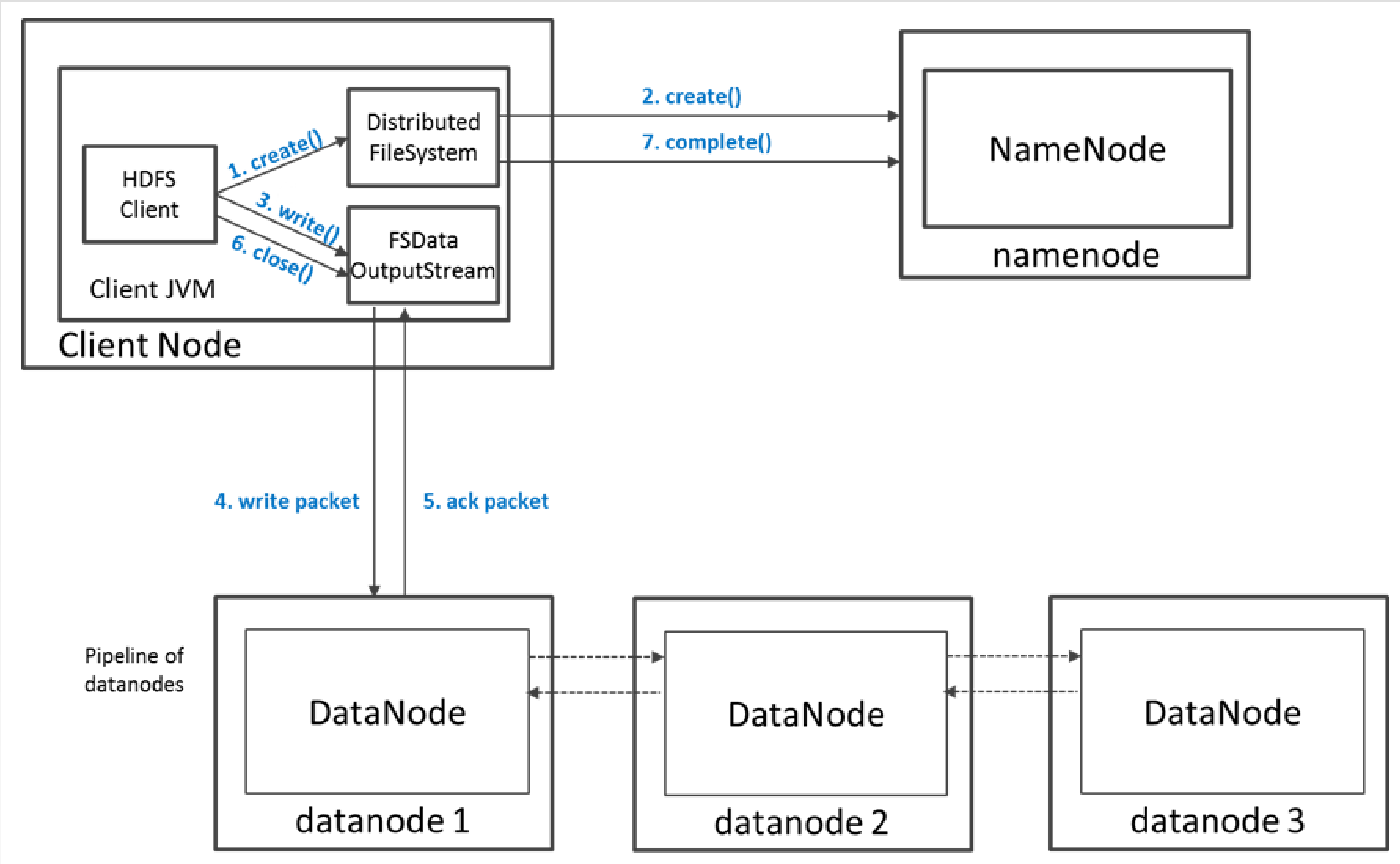
- The NameNode makes all decisions regarding replication of blocks.



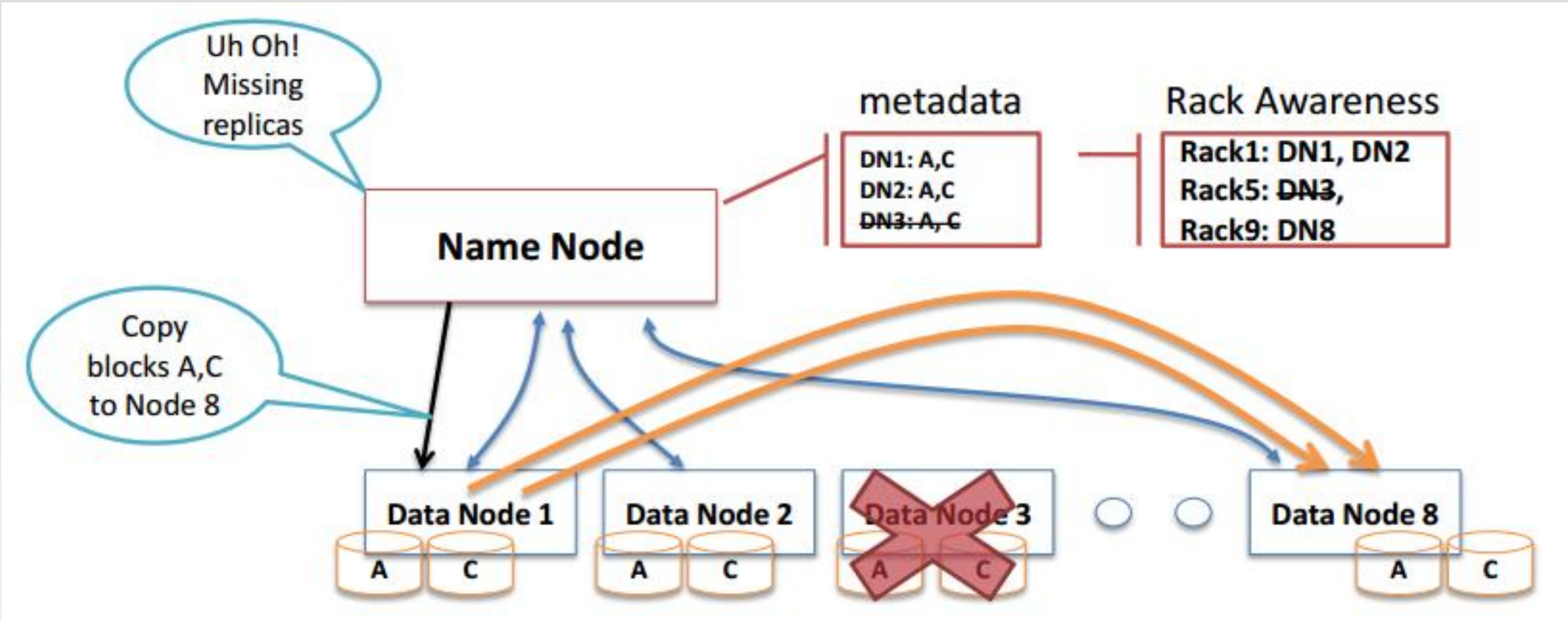
# File Read Data Flow in HDFS



# File Write Data Flow in HDFS



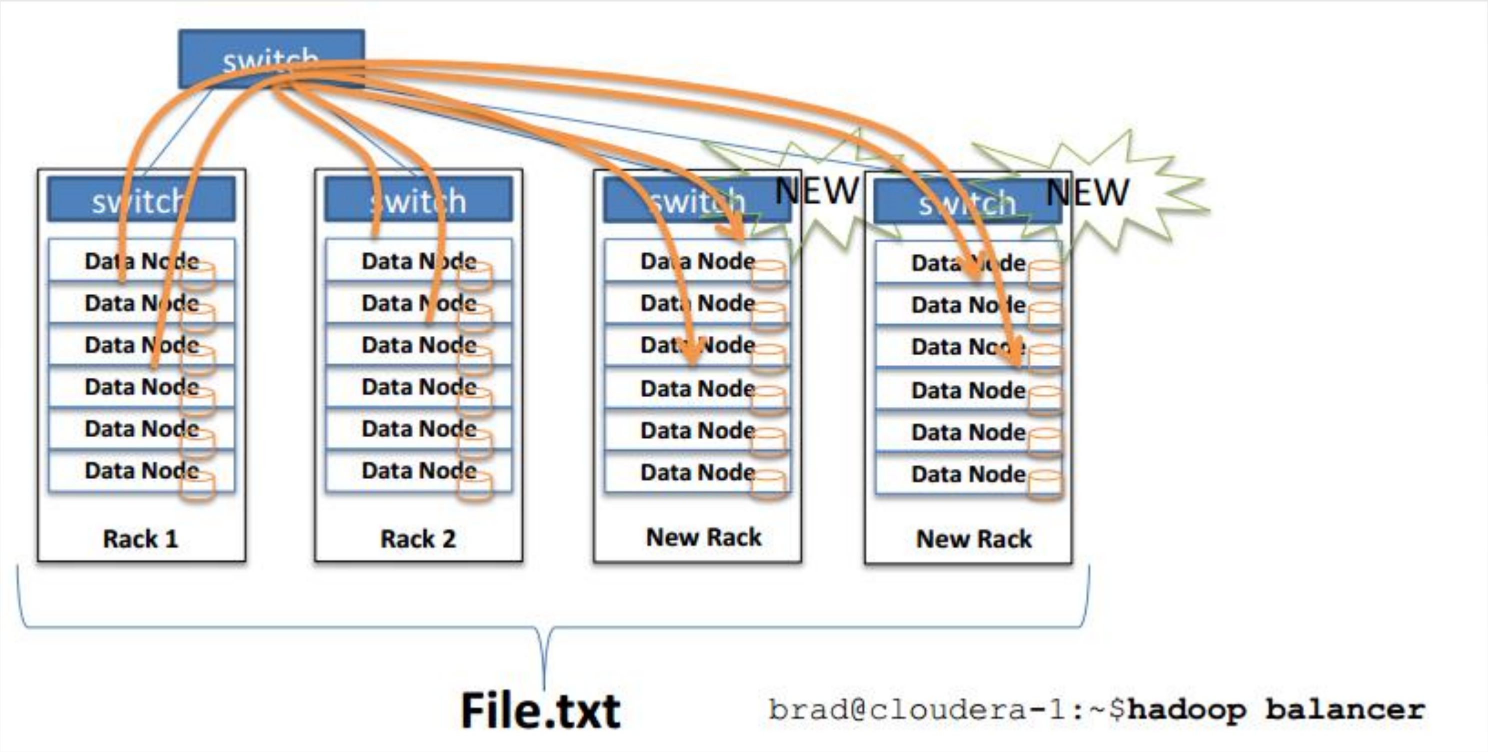
# Replication Engine




- NameNode detects DataNode failures
  - Missing Heartbeats signify lost Nodes
  - NameNode consults metadata, finds affected data
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

# Cluster Rebalancing

- Goal: % disk full on DataNodes should be similar
  - Usually run when new DataNodes are added
  - Rebalancer is throttled to avoid network congestion
  - Does not interfere with MapReduce or HDFS
  - Command line tool



# Fault tolerance

- Failure is the norm rather than exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

# Metadata Disk Failure

- FsImage and EditLog are central data structures of HDFS. A corruption of these files can cause a HDFS instance to be non-functional.
  - A NameNode can be configured to maintain multiple copies of the FsImage and EditLog
  - Multiple copies of the FsImage and EditLog files are updated synchronously

# HDFS Erasure Coding

- Replication is expensive – the default 3x replication scheme in HDFS has 200% overhead in storage space and other resources.

- Therefore, a natural improvement is to use Erasure Coding (EC) in place of replication, which provides the same level of fault-tolerance with much less storage space.
  - Erasure Coding transforms a message of *k* symbols into a longer message with *n* symbols such that the original message can be recovered from a subset of the *n* symbols.
  - In typical Erasure Coding (EC) setups, the storage overhead is no more than 50%. Replication factor of an EC file is meaningless. It is always 1 and cannot be changed via -setrep command.

# Unique features of HDFS

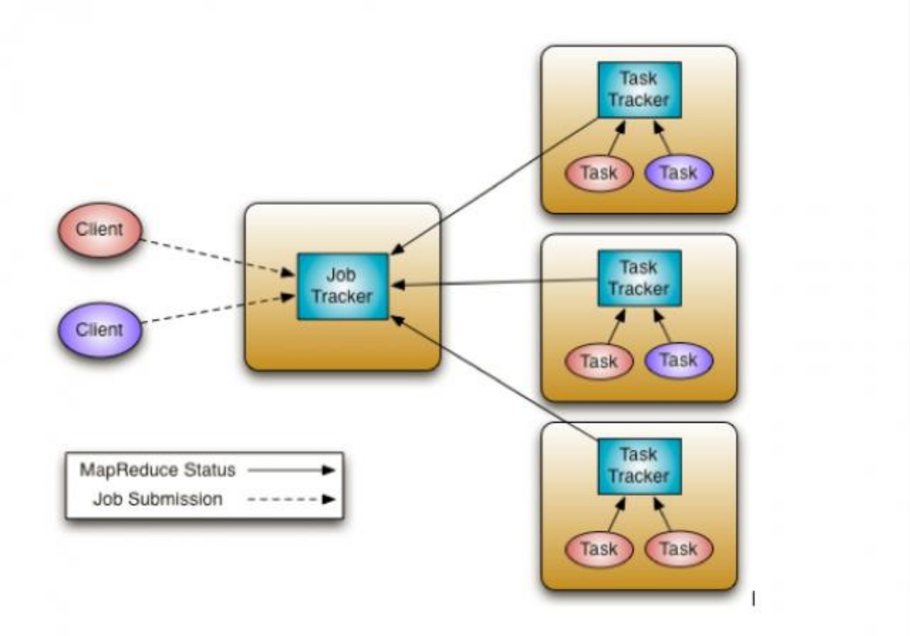
- HDFS has a bunch of unique features that make it ideal for distributed systems:
  - Failure tolerant - data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).
  - Scalability - data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes
  - Space - need more disk space? Just add more DataNodes and re-balance
  - Industry standard - Other distributed applications are built on top of HDFS (HBase, MapReduce)

- HDFS is designed to process large data sets with write-once-read-many semantics, it is not for low latency access

# Part 2: YARN

# Why YARN

- In Hadoop version 1, MapReduce performed both processing and resource management functions.
  - It consisted of a Job Tracker which was the single master. The Job Tracker allocated the resources, performed scheduling and monitored the processing jobs.
  - It assigned map and reduce tasks on a number of subordinate processes called the Task Trackers. The Task Trackers periodically reported their progress to the Job Tracker.



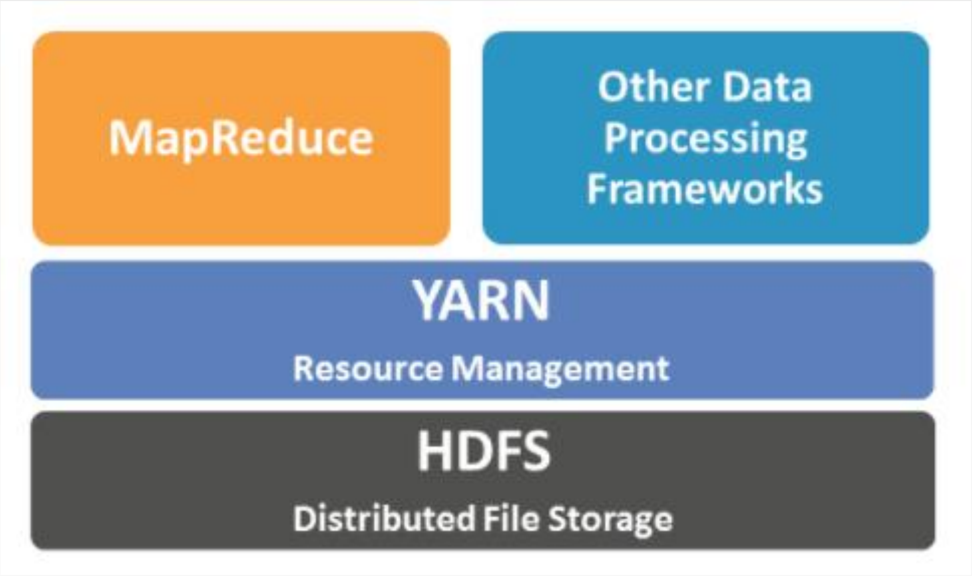
# What is YARN

- YARN - “**Y**et **A**nother **R**esource **N**egotiator”
  - The resource management layer of Hadoop, introduced in Hadoop 2.x
  - monitors and manages workloads, maintains a multi-tenant environment, manages the high availability features of Hadoop, and implements security controls

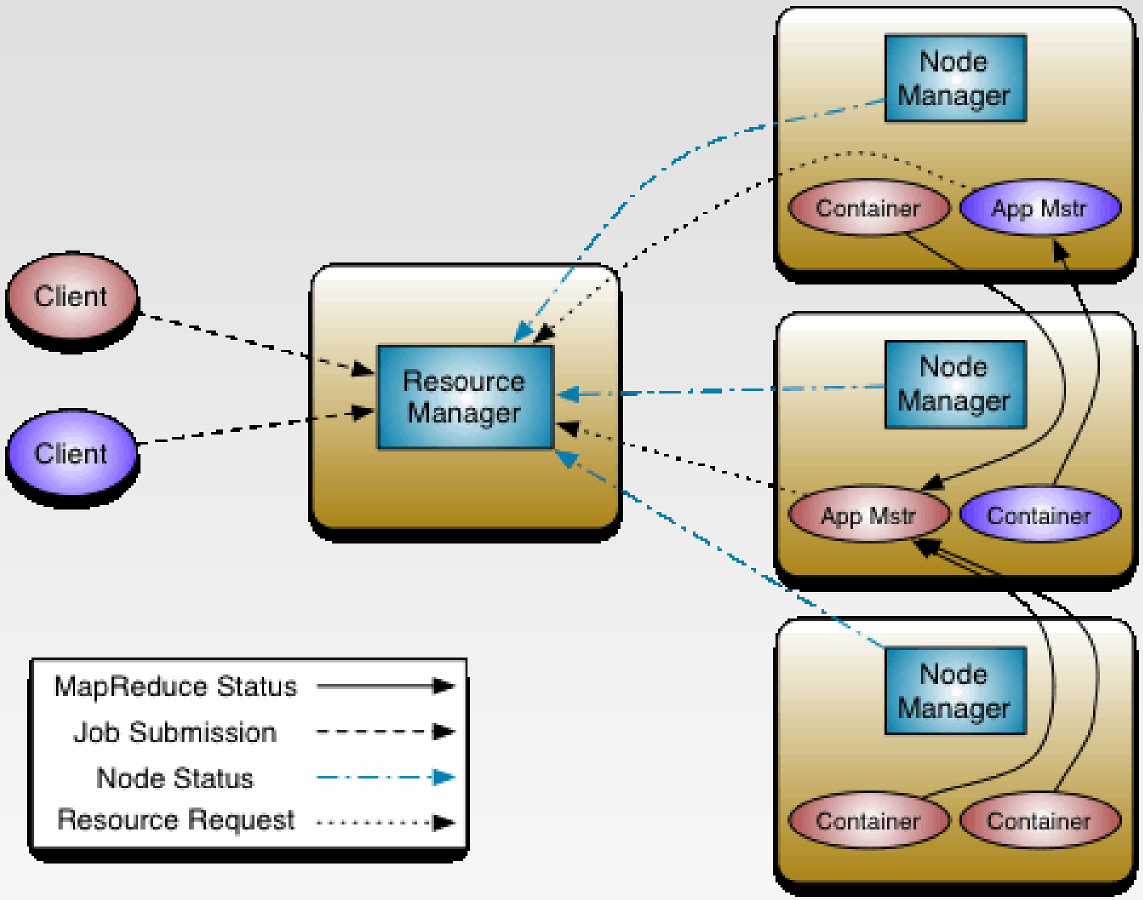
- Motivation:
  - Flexibility - Enabling data processing model more than MapReduce
  - Efficiency - Improving performance and QoS
  - Resource Sharing - Multiple workloads in cluster

# What is YARN

- YARN was introduced in Hadoop version 2.0 in the year 2012 by Yahoo and Hortonworks.
- The basic idea behind YARN is to relieve MapReduce by taking over the responsibility of Resource Management and Job Scheduling.
- YARN enabled the users to perform operations as per requirement by using a variety of tools like Spark for real-time processing, Hive for SQL, HBase for NoSQL and others.



# YARN Framework



# YARN Components

- ResourceManager
  - Arbitrates resources among all the applications in the system
- ApplicationMaster
  - A framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks
- NodeManager
  - The per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager
- Container
  - Unit of allocation incorporating resource elements such as memory, cpu, disk, network etc, to execute a specific task of the application

# Resource Manager

- It is the ultimate authority in resource allocation.
- On receiving the processing requests, it passes parts of requests to corresponding node managers accordingly, where the actual processing takes place.
- It is the arbitrator of the cluster resources and decides the allocation of the available resources for competing applications.
- Optimizes the cluster utilization like keeping all resources in use all the time against various constraints such as capacity guarantees, fairness, and SLAs.
- It has two major components:
  - a) Scheduler
  - b) Application Manager

# Scheduler

- The scheduler is responsible for allocating resources to the various running applications subject to constraints of capacities, queues etc.
- It is called a pure scheduler in ResourceManager, which means that it does not perform any monitoring or tracking of status for the applications.
- If there is an application failure or hardware failure, the Scheduler does not guarantee to restart the failed tasks.
- Performs scheduling based on the resource requirements of the applications.
- It has a pluggable policy plug-in, which is responsible for partitioning the cluster resources among the various applications. There are two such plug-ins: Capacity Scheduler and Fair Scheduler, which are currently used as Schedulers in ResourceManager.

# Application Manager

- It is responsible for accepting job submissions.
- Negotiates the first container from the ResourceManager for executing the application specific ApplicationMaster.
- Manages running the ApplicationMasters in a cluster and provides service for restarting the ApplicationMaster container on failure.

# Node Manager

- It takes care of individual nodes in a Hadoop cluster and manages user jobs and workflow on the given node.
- It registers with the ResourceManager and sends heartbeats with the health status of the node.
- Its primary goal is to manage application containers assigned to it by the resource manager.
- It keeps up-to-date with the ResourceManager.
- Application Master requests the assigned container from the NodeManager by sending it a Container Launch Context(CLC) which includes everything the application needs in order to run. The NodeManager creates the requested container process and starts it.
- Monitors resource usage (memory, CPU) of individual containers.
- Performs Log management.
- It also kills the container as directed by the ResourceManager.

# Application Master

- An application is a single job submitted to the framework. Each such application has a unique Application Master associated with it which is a framework specific entity.
- It is the process that coordinates an application’s execution in the cluster and also manages faults.
- Its task is to negotiate resources from the ResourceManager and work with the NodeManager to execute and monitor the component tasks.
- It is responsible for negotiating appropriate resource containers from the ResourceManager, tracking their status and monitoring progress.
- Once started, it periodically sends heartbeats to the ResourceManager to affirm its health and to update the record of its resource demands.

# Container

- It is a collection of physical resources such as RAM, CPU cores, and disks on a single node.
- YARN containers are managed by a container launch context which is container life-cycle(CLC). This record contains a map of environment variables, dependencies stored in a remotely accessible storage, security tokens, payload for NodeManager services and the command necessary to create the process.
- It grants rights to an application to use a specific amount of resources (memory, CPU etc.) on a specific host.

# Application Workflow in YARN

- Execution Sequence
  - 1. A client program submits the application
  - 2. ResourceManager allocates a specified container to start the ApplicationMaster
  - 3. ApplicationMaster, on boot-up, registers with ResourceManager
  - 4. ApplicationMaster negotiates with ResourceManager for appropriate resource containers
  - 5. On successful container allocations, ApplicationMaster contacts NodeManager to launch the container
  - 6. Application code is executed within the container, and then ApplicationMaster is responded with the execution status
  - 7. During execution, the client communicates directly with ApplicationMaster or ResourceManager to get status, progress updates etc.
  - 8. Once the application is complete, ApplicationMaster unregisters with ResourceManager and shuts down, allowing its own container process

# Part 3: MapReduce

# What is MapReduce

- Origin from Google, [OSDI'04]
  - MapReduce: Simplified Data Processing on Large Clusters
  - Jeffrey Dean and Sanjay Ghemawat
- Programming model for parallel data processing
- Hadoop can run MapReduce programs written in various languages: e.g. Java, Ruby, Python, C++
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in a distributed manner
  - Offers high availability

# Motivation for MapReduce

- Typical big data problem challenges:
  - How do we break up a large problem into smaller tasks that can be executed in parallel?
  - How do we assign tasks to workers distributed across a potentially large number of machines?
  - How do we ensure that the workers get the data they need?
  - How do we coordinate synchronization among the different workers?
  - How do we share partial results from one worker that is needed by another?
  - How do we accomplish all of the above in the face of software errors and hardware faults?

# Motivation for MapReduce

- There was need for an abstraction that hides many system-level details from the programmer.

- MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a ***scalable***, ***robust***, and ***efficient*** manner.

- MapReduce separates the ***what*** from the ***how***

# Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each **Map**
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output **Reduce**

Key idea: provide a functional abstraction for these two operations

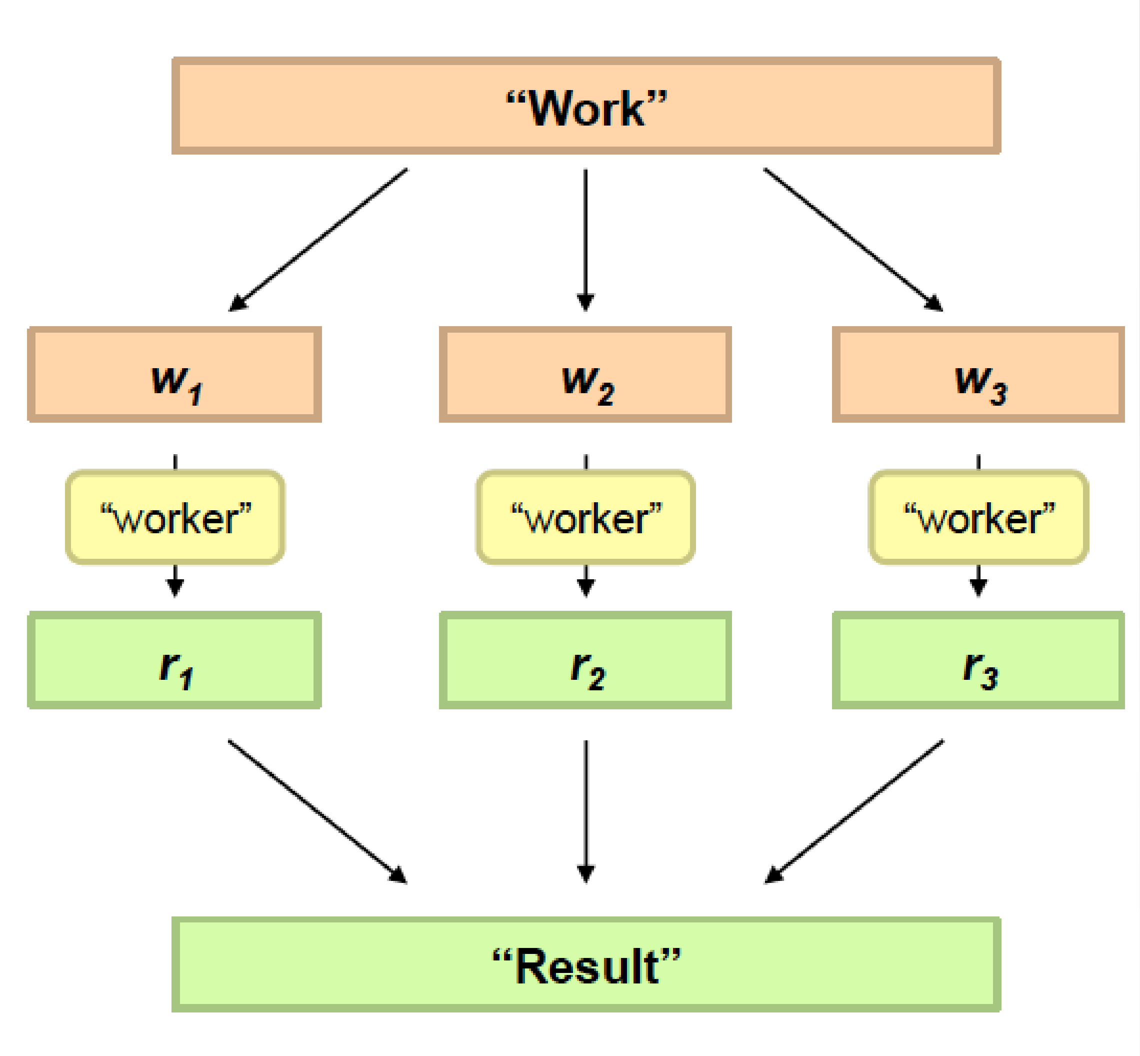
# The Idea of MapReduce

- Inspired by the map and reduce functions in functional programming
- We can view map as a transformation over a dataset
  - This transformation is specified by the function *f*
  - Each functional application happens in isolation
  - The application of *f* to each element of a dataset can be parallelized in a straightforward manner
- We can view reduce as an aggregation operation
  - The aggregation is defined by the function *g*
  - Data locality: elements in the list must be “brought together”
  - If we can group elements of the list, also the reduce phase can proceed in parallel
- The framework coordinates the map and reduce phases:
  - Grouping intermediate results happens in parallel

# Everything Else?

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (HDFS)
- You don't know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

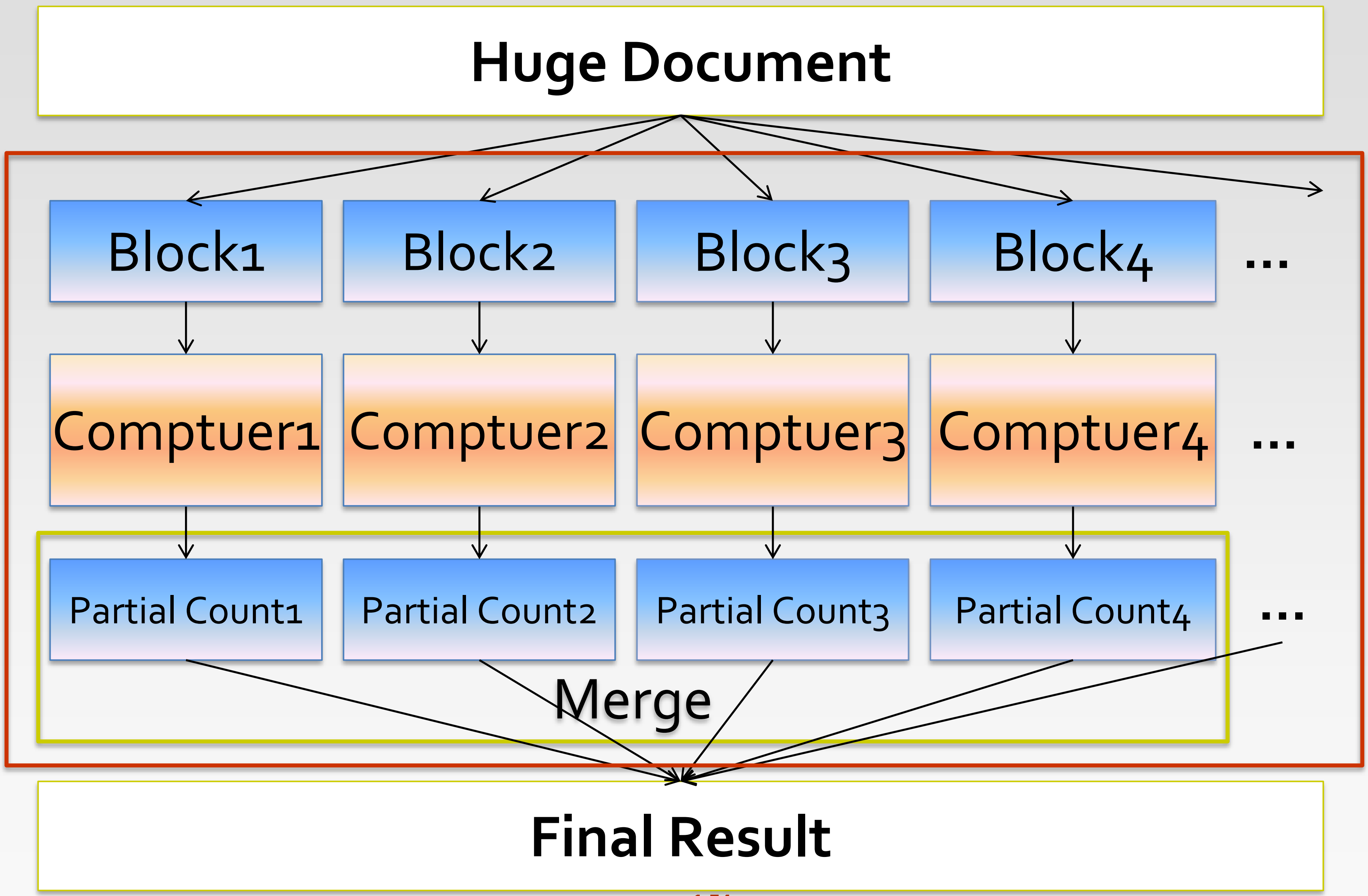
# Philosophy to Scale for Big Data Processing




Divide Work

Combine Results

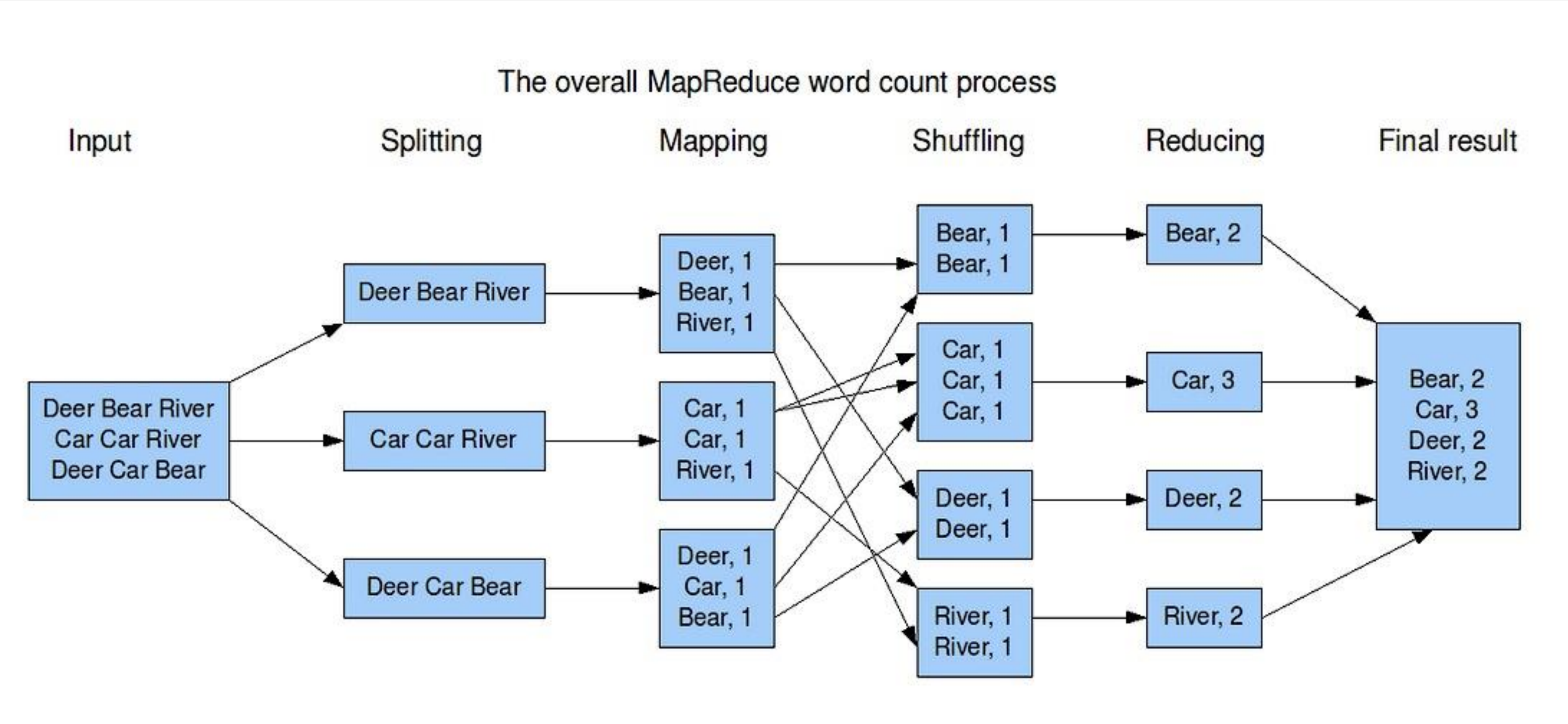
# Distributed Word Count



# Distributed Word Count

- Challenges?
  - Where to store the huge document dataset?
  - How to split the dataset into different blocks?
    - How many blocks?
    - The size of each block?
  - What can we do if one node lost the data it received?
  - What can we do if one node cannot be connected?
  - … …

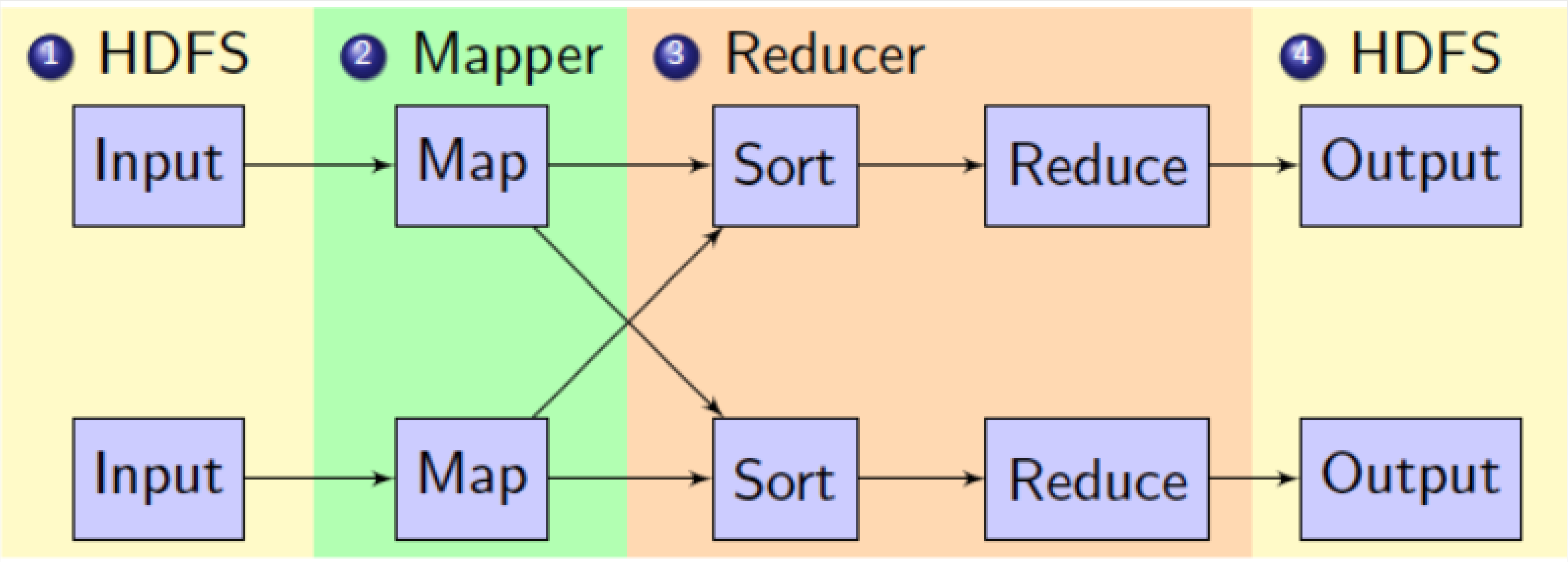
# MapReduce Example - WordCount




- Hadoop MapReduce is an implementation of MapReduce
  - MapReduce is a computing paradigm (Google)
  - Hadoop MapReduce is an open-source software

# Hadoop MapReduce Brief Data Flow

- 1. Mappers read from HDFS
- 2. Map output is partitioned by key and sent to Reducers
- 3. Reducers sort input by key
- 4. Reduce output is written to HDFS
- Intermediate results are stored on local FS of Map and Reduce workers



# End of Chapter 1.1

# References


- HDFS Architecture. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html
- Understanding Hadoop Clusters and the Network. https://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/
- YARN tutorial. https://www.edureka.co/blog/hadoop-yarn-tutorial/