Cloud Programming and Software Environments

Map Reduce and Hadoop

Adapted from Kai Hwang, University of Southern California
with additions from
Matei Zaharia, EECS, UC Berkeley

Overview

- Process flow in MapReduce
- Motivation for Programming Paradigm
- MapReduce
  - MapReduce Operational Steps
- Twister : Iterative MapReduce
- Hadoop
  - HDFS Architecture
  - Read and Write Operations in HDFS
  - MapReduce Architecture in Hadoop
  - Running a job in Hadoop
  - Challenges

Parallel Computing and Programming Paradigms
Process Flow in MapReduce

- **Partitioning**
  - Computation Partitioning: This splits a given job or a program into smaller tasks. Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently.
  - Data Partitioning: This splits the input or intermediate data into smaller pieces.
- **Mapping (Resource Allocator):** This process aims to appropriately assign smaller parts of data and pieces of program to be run simultaneously on different workers and is usually handled by resource allocators in the system
- **Synchronization:** Because different workers may perform different tasks, synchronization and coordination among workers is necessary so that race conditions are prevented and data dependency among different workers is properly managed
- **Communication:** Because data dependency is one of the main reasons for communication among workers, communication is always triggered when the intermediate data is sent to workers
- **Scheduling (Resource Allocator and Scheduler):** For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers.

Motivation for Programming Paradigm

- To improve productivity for programmers
- To decrease programs time to market
- To leverage underlying resources more efficiently
- To increase system throughput
- To support higher levels of abstraction
  - MapReduce, Hadoop and Dryad are parallel and distributed programming models.

What is MapReduce?

- Simple data-parallel programming model
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in distributed manner
  - Offers high availability
- Pioneered by Google
  - Processes 20 petabytes of data per day
- Popularized by open-source Hadoop project
  - Used at Yahoo!, Facebook, Amazon, …

What is MapReduce used for?

- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - "Web map" powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

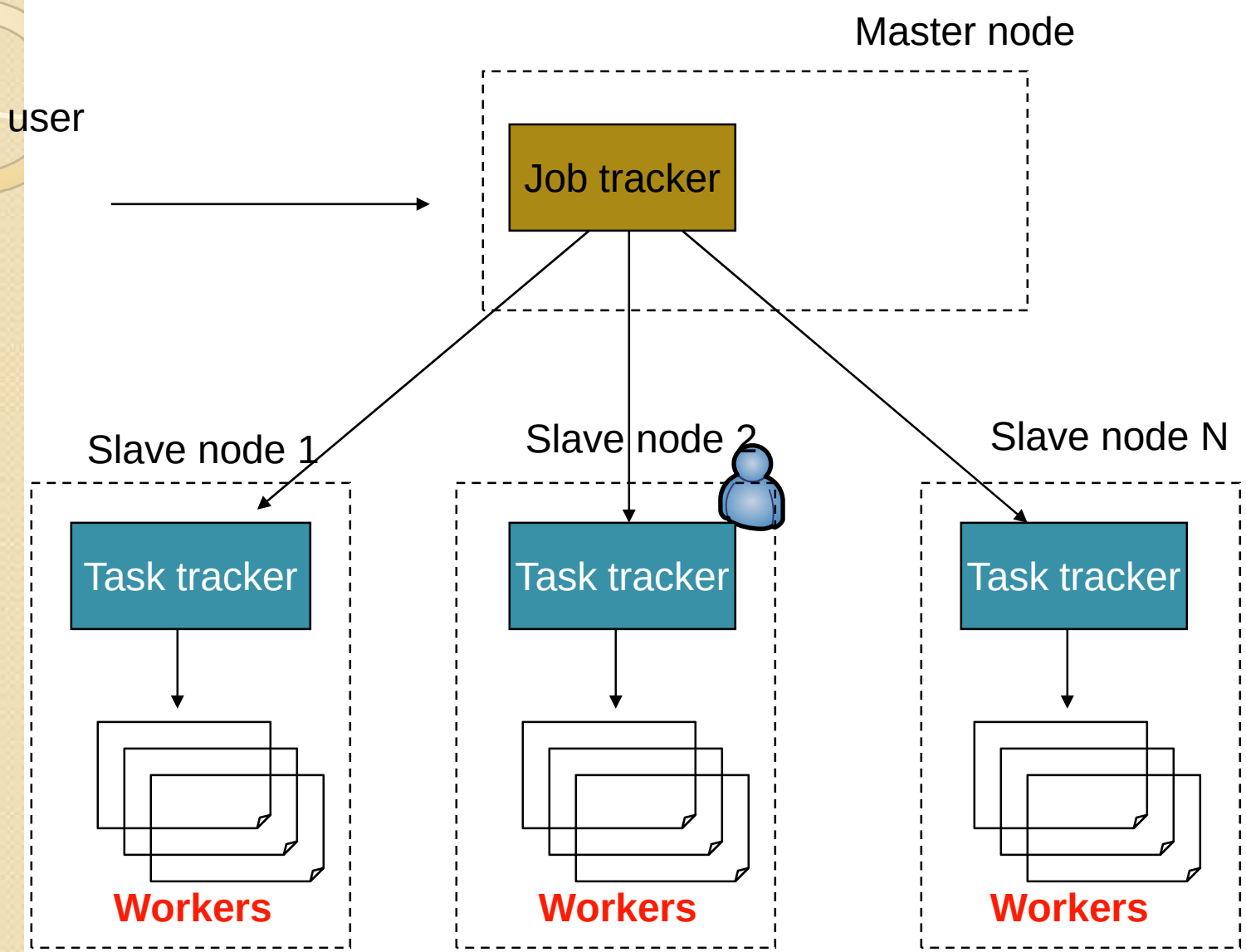
Motivation: Large Scale Data Processing

- Many tasks composed of processing lots of data to produce lots of other data
- Want to use hundreds or thousands of CPUs
  ... but this needs to be easy!
- **Map Reduce provides**
  - User-defined functions
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

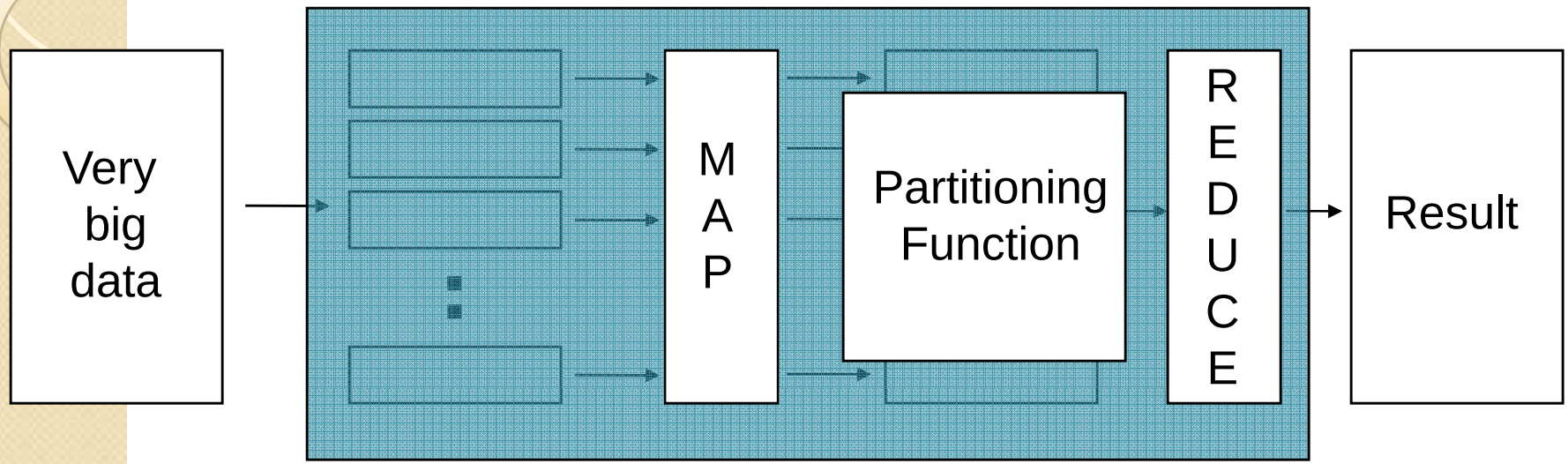
What is MapReduce used for?

- In research:
  - Astronomical image analysis (Washington)
  - Bioinformatics (Maryland)
  - Analyzing Wikipedia conflicts (PARC)
  - Natural language processing (CMU)
  - Particle physics (Nebraska)
  - Ocean climate simulation (Washington)
  - <Your application here>

Map Reduce Architecture



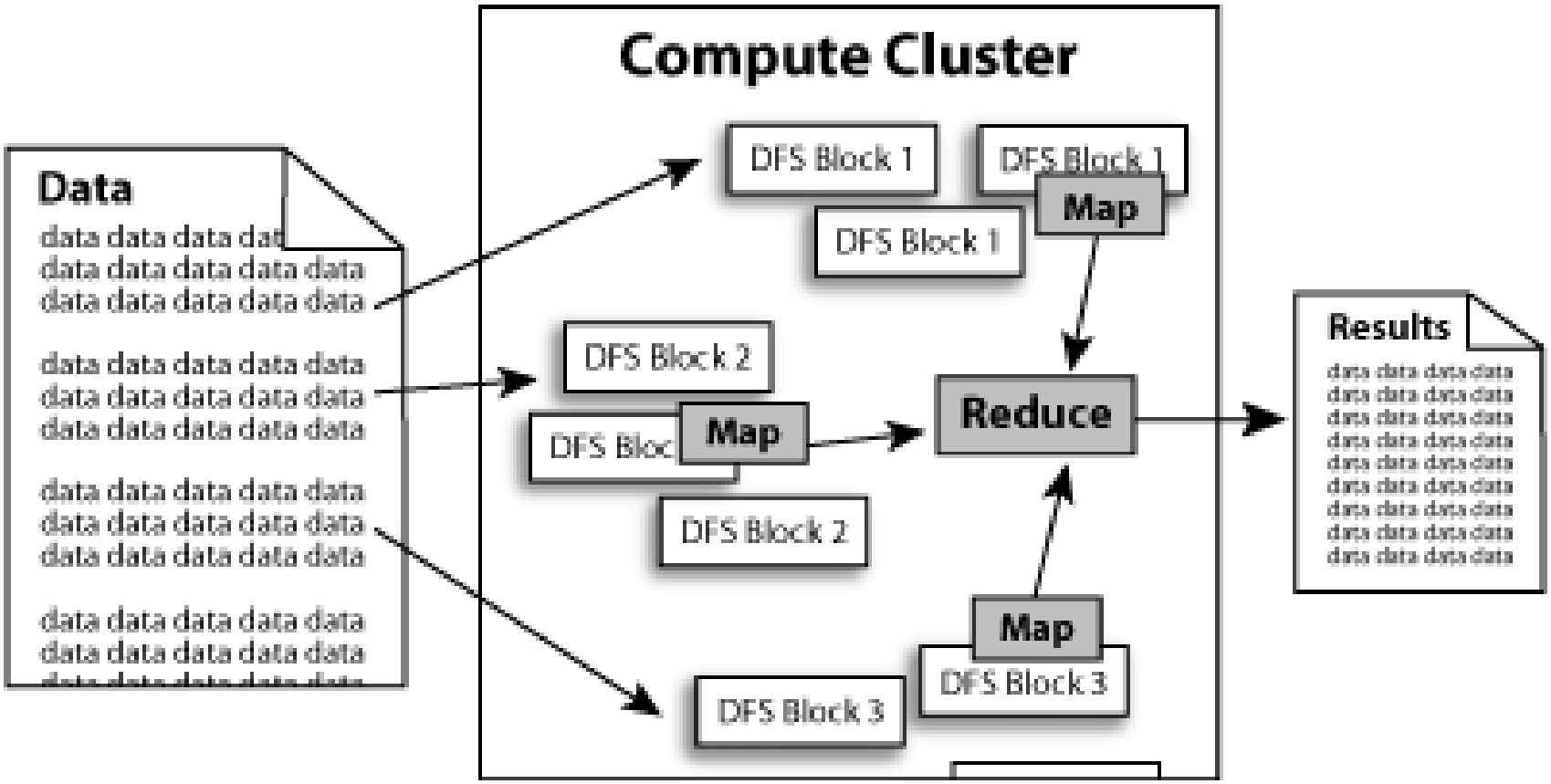
Map+Reduce



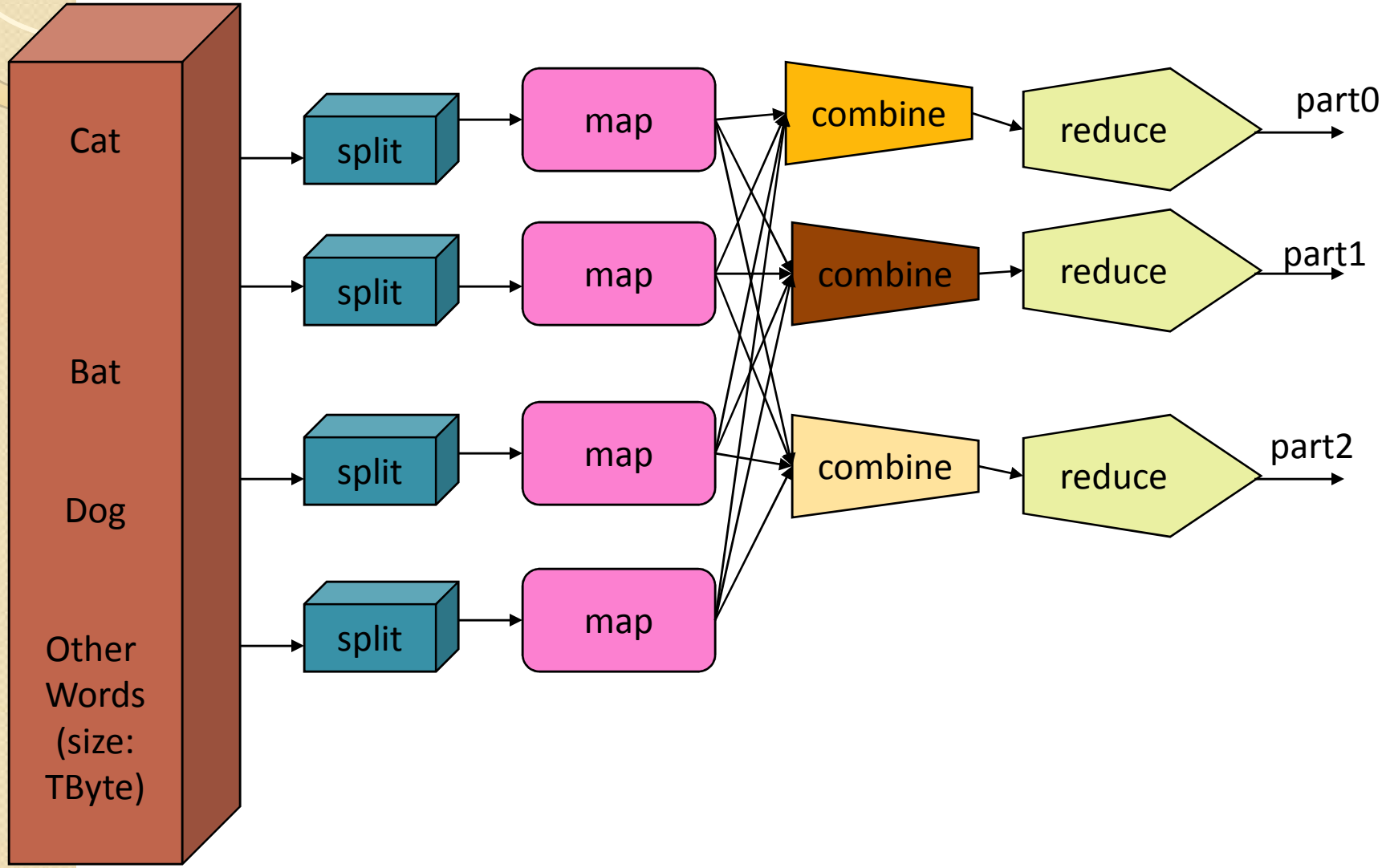
- Map:
  - Accepts *input* key/value pair
  - Emits *intermediate* key/value pair
- Reduce :
  - Accepts *intermediate* key/value* pair
  - Emits *output* key/value pair

Functions in the Model

- Map
  - Process a key/value pair to generate intermediate key/value pairs
- Reduce
  - Merge all intermediate values associated with the same key
- Partition
  - By default : `hash(key) mod R`
  - Well balanced


Compute Cluster
Data
data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
data data data data data
DFS Block 1
DFS Block 1
DFS Block 1
Map
DFS Block 2
DFS Bloc
Map
Reduce
DFS Block 2
Map
DFS Block 3
DFS Block 3
Results
data data data data
data data data data
data data data data
data data data data
data data data data
data data data data
data data data data
data data data data
data data data data

MapReduce


Cat
Bat
Dog
Other Words (size: TByte)
split
split
split
split
map
map
map
map
combine
combine
combine
reduce
reduce
reduce
part0
part1
part2

A Simple Example

- Counting words in a large set of documents

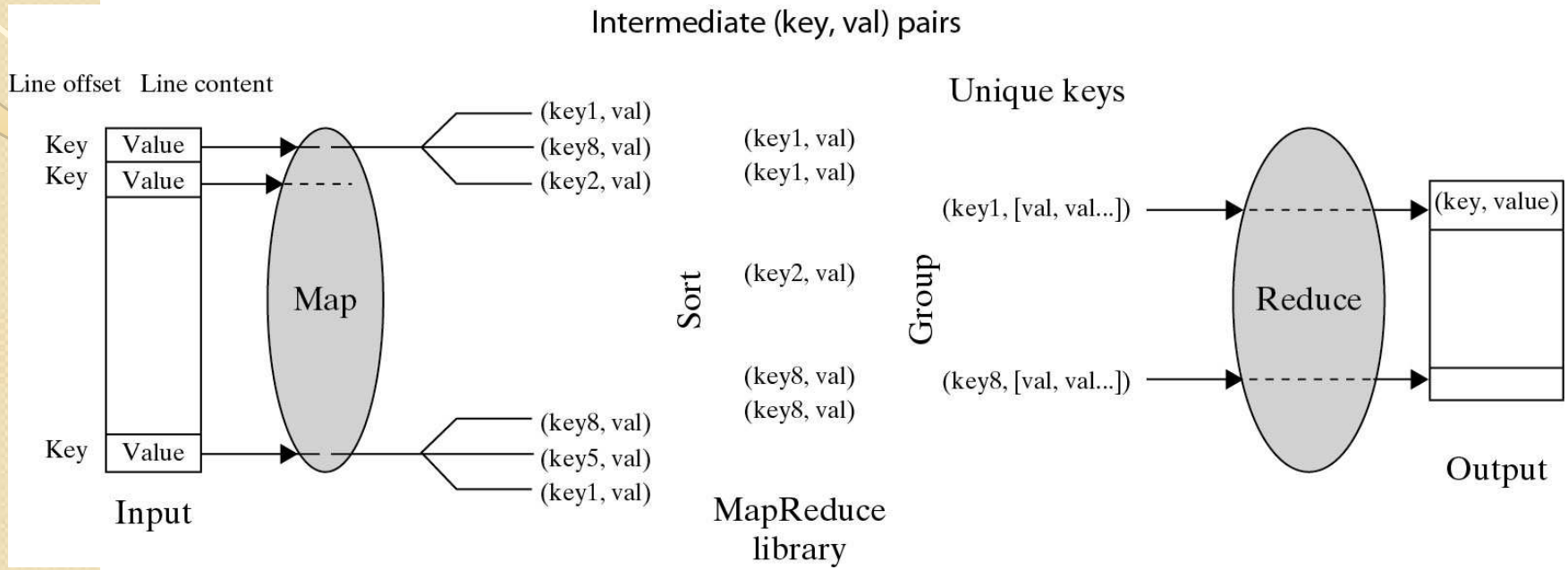
```
map(string value)
    //key: document name
    //value: document contents
    for each word w in value
        EmitIntermediate(w, "1");
```

The map function emits each word *w* plus an associated count of occurrences (just a "1" is recorded in this pseudo-code)

```
reduce(string key, iterator values)
    //key: word
    //values: list of counts
    int results = 0;
    for each v in values
        result += ParseInt(v);
    Emit(AsString(result));
```

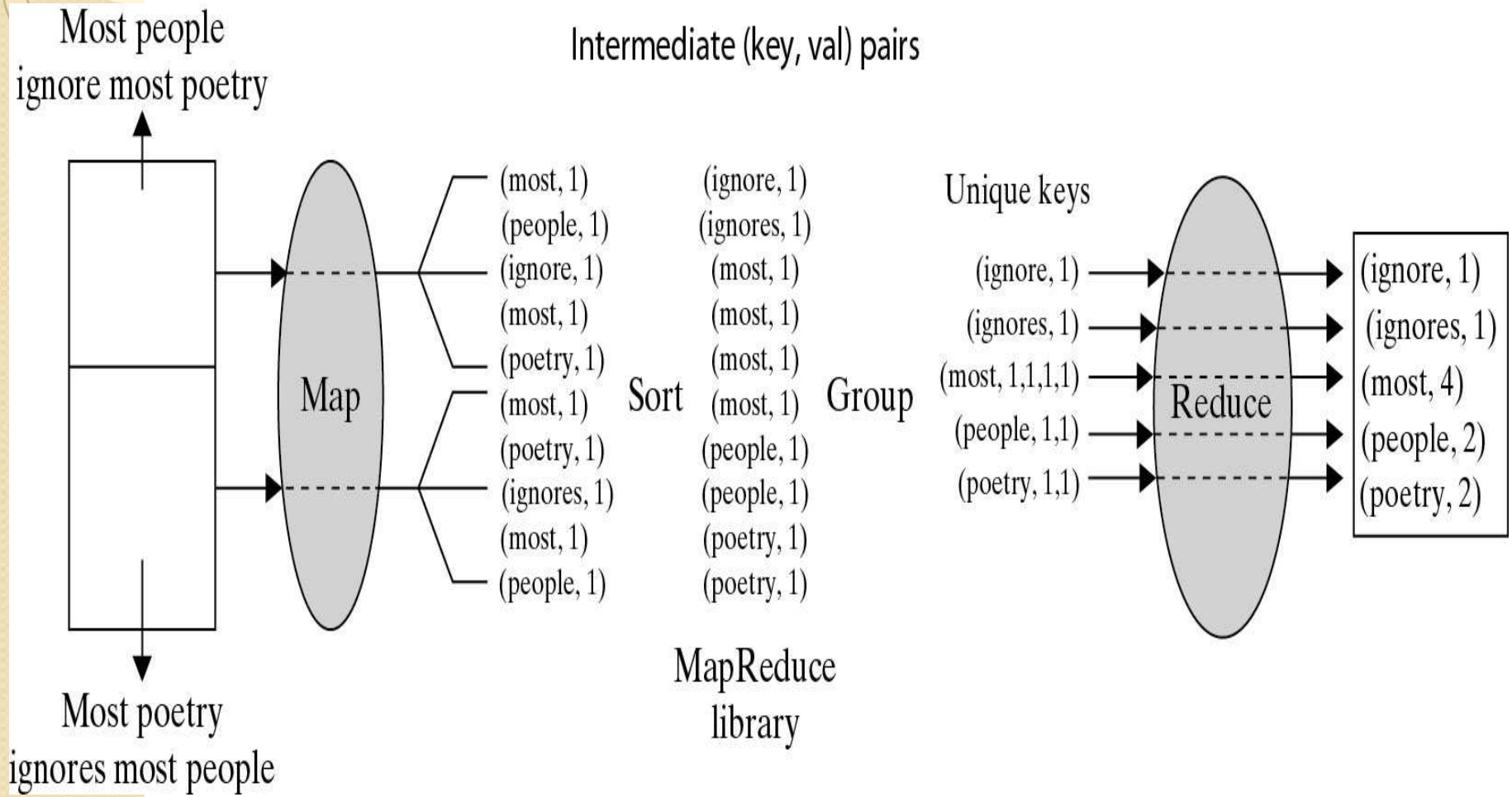
The reduce function sums together all counts emitted for a particular word

Logical Data Flow in 5 Processing Steps in MapReduce Process



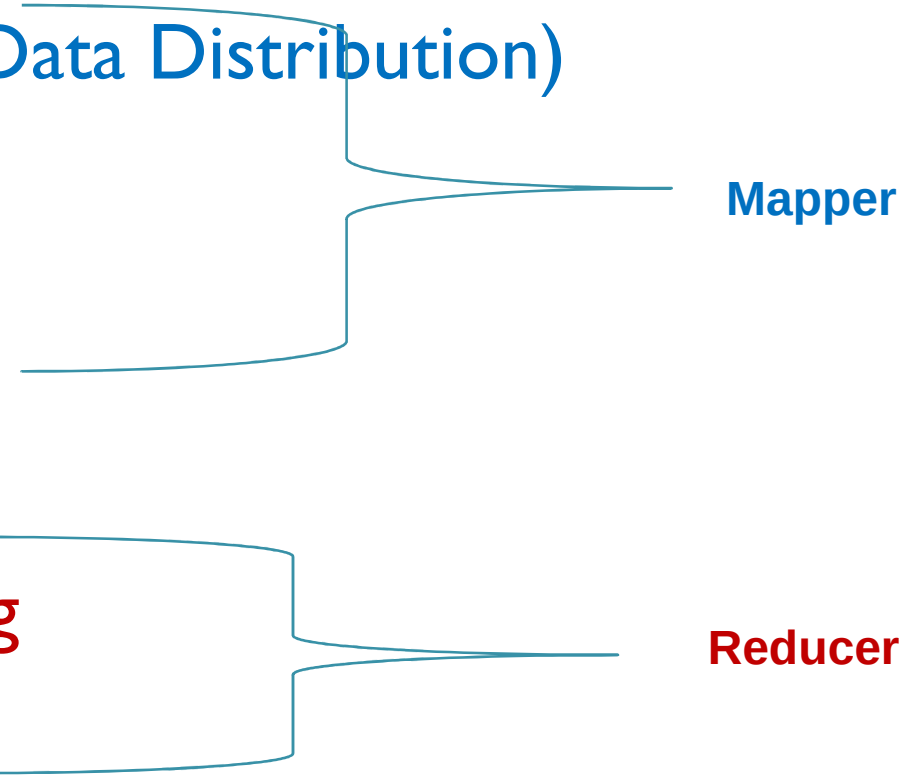
(Key,Value) Pairs are generated by the Map function over multiple available Map Workers (VM instances). These pairs are then sorted and group based on key ordering. Different key-groups are then processed by multiple Reduce Workers in parallel.

A Word Counting Example on <Key, Count> Distribution



Actual MapReduce Data and Control Flow

- Data Partitioning
- Computation Partitioning
- Determining Master and workers
- Reading Input data (Data Distribution)
- Map Function
- Combiner Function
- Partitioning Function
- Synchronization
- Communication
- Sorting and Grouping
- Reduce Function



Actual MapReduce Data and Control Flow

- **Data Partitioning**: The MapReduce library splits the input data (files), already stored in GFS, into M pieces that also correspond to the number of map tasks.
- **Computation Partitioning**: MapReduce library only generates copies of a user program (e.g., by a fork system call) containing the Map and the Reduce functions, distributes them, and starts them up on a number of available computation engines.
- **Determining Master and workers:** The MapReduce architecture is based on a master-worker model. Therefore, one of the copies of the user program becomes the master and the rest become workers. The master picks idle workers, and assigns the map and reduce tasks to them.

Actual MapReduce Data and Control Flow

- **Mapper Functions**
  - Reading Input data (Data Distribution) : Each map worker reads its corresponding portion of the input data split and sends it to its Map function
  - Map Function : Each Map function receives the input data split as a set of (key, value) pairs to process and produce the intermediated (key, value) pairs.
  - Combiner Function: The Combiner function merges the local data of each map worker before sending it over the network to effectively reduce its communication costs
  - Partitioning Function : The intermediate (key, value) pairs with identical keys are grouped together because all values inside each group should be processed by only one Reduce function to generate the final result. Intermediate (key, value) pairs are partitioned into R regions, equal to the number of reduce tasks by Partitioning Function. A Partitioning function could simply be a hash function (e.g., Hash(key) mod R) that forwards the data into particular regions.

MapReduce Partitioning Function

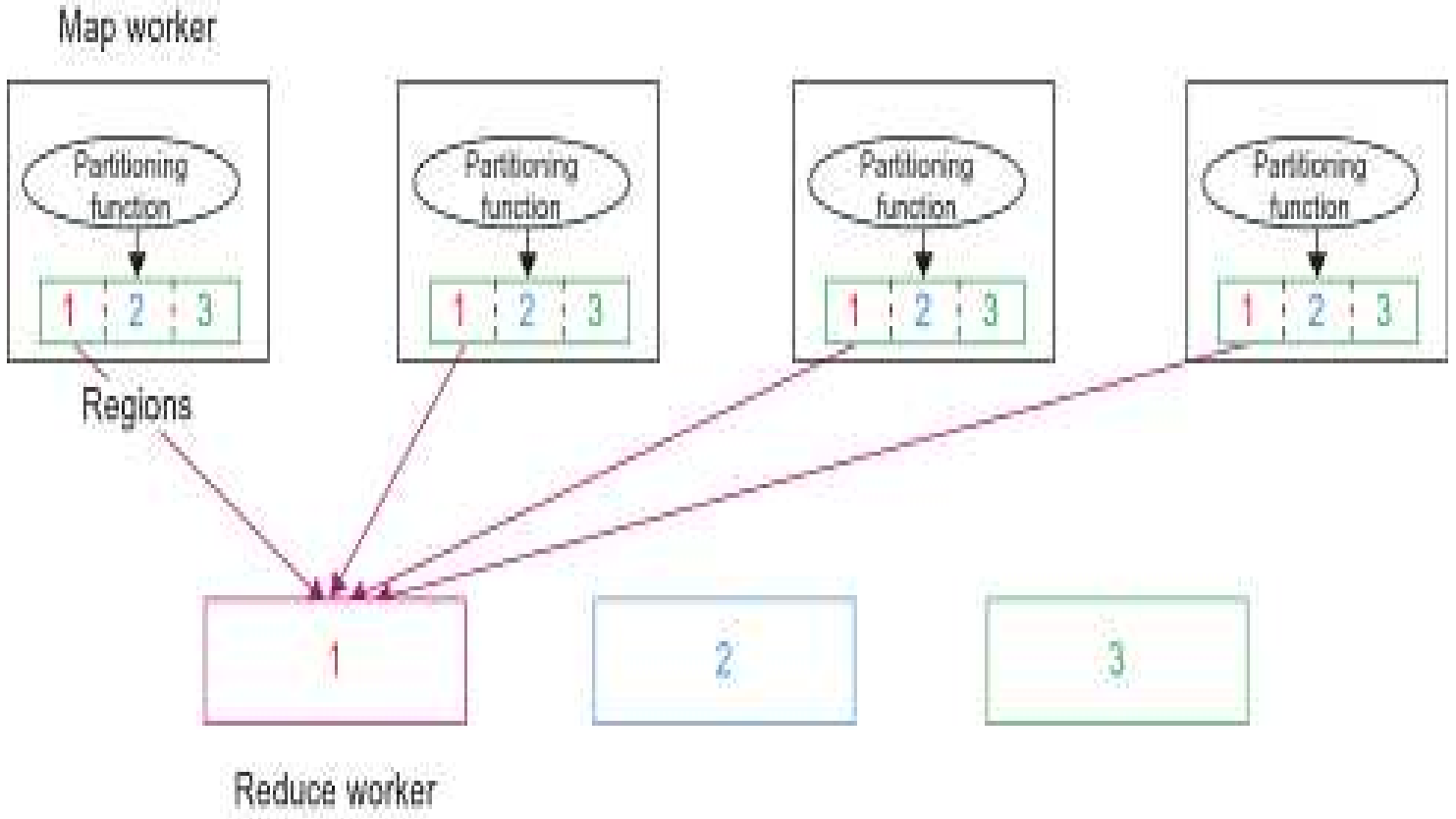


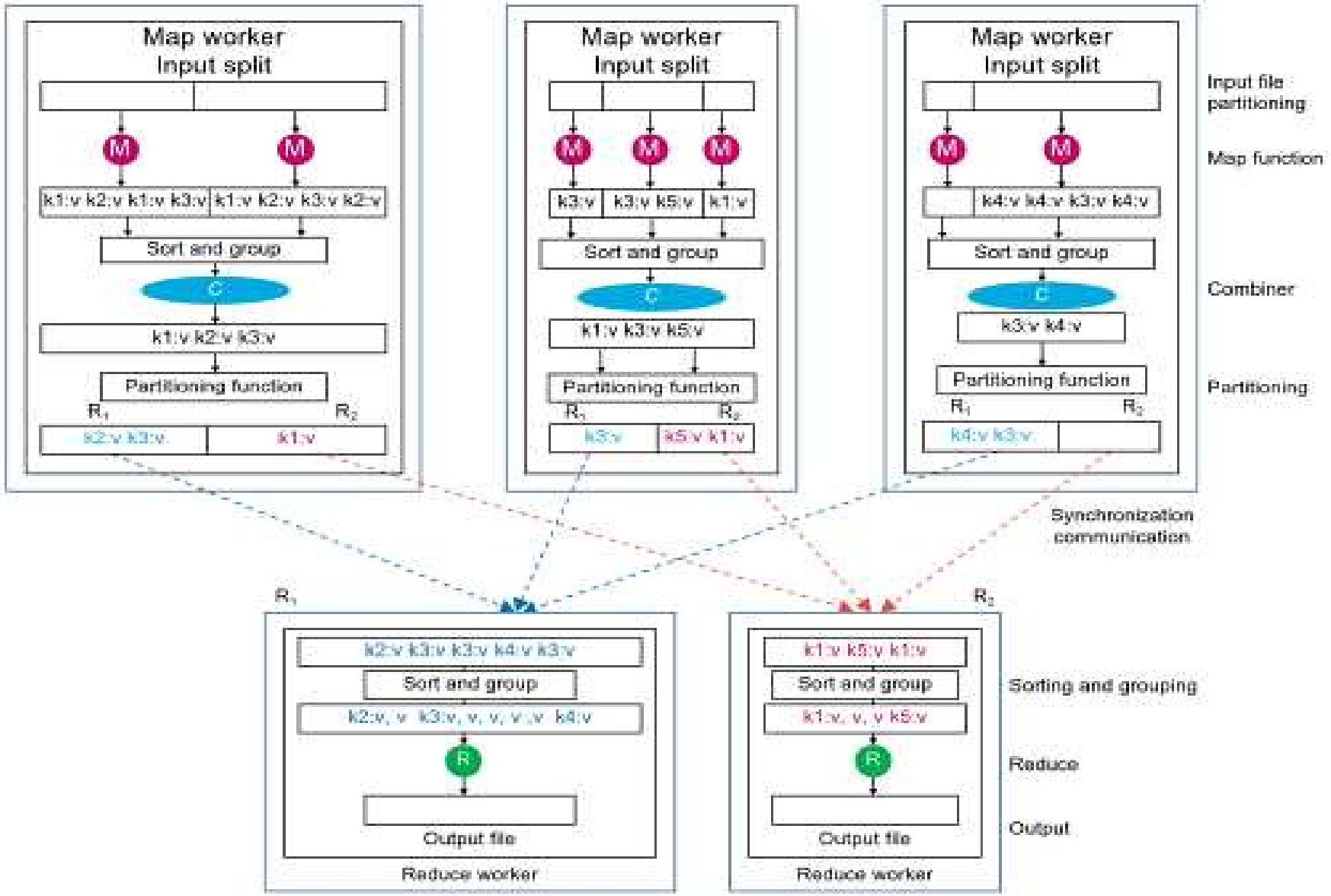
FIGURE 6.4

Use of MapReduce *partitioning* function to link the Map and Reduce workers.

Actual MapReduce Data and Control Flow

- **Synchronization**: MapReduce applies a simple synchronization policy to coordinate map workers with reduce workers, in which the communication between them starts only when all map tasks finish.
- **Communication**: Reduce worker i, already notified of the location of region i of all map workers, uses a remote procedure call to read the data from the respective region of all map workers.
- **Reduce Function**
  - Sorting and Grouping: Reduce worker groups intermediate (key, value) pairs by sorting the data based on their keys, followed by grouping all occurrences of identical keys.
  - Reduce Function: The reduce worker iterates over the grouped (key, value) pairs, and for each unique key, it sends the key and corresponding values to the Reduce function. Reduce function processes its input data and stores the output results in predetermined files in the user's program.

Data Flow of MapReduce



Points need to be emphasized

- No *reduce* can begin until *map* is complete
- Master must communicate locations of intermediate files
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely re-run
- MapReduce library does most of the hard work for us!

Locality issue

- Master scheduling policy
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (== GFS block size)
  - Map tasks scheduled so that GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate

Fault Tolerance

- Reactive way
  - Worker failure
    - Heartbeat, Workers are periodically pinged by master
      - NO response = failed worker
    - If the processor of a worker fails, the tasks of that worker are reassigned to another worker.
  - Master failure
    - Master writes periodic checkpoints
    - Another master can be started from the last checkpointed state
    - If eventually the master dies, the job will be aborted

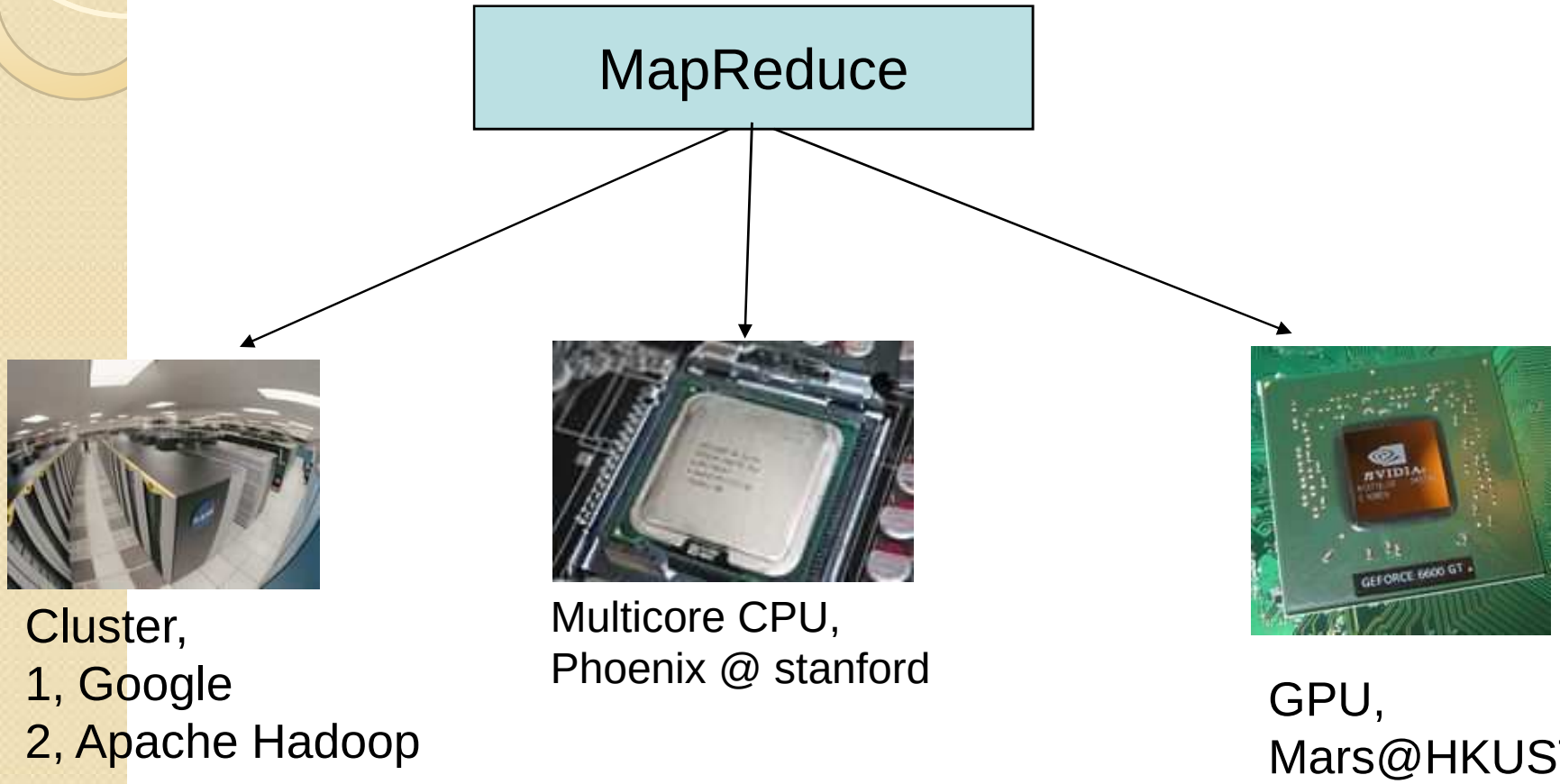
Fault Tolerance

- Proactive way (**Redundant Execution**)
  - When computation almost done, reschedule in-progress tasks
  - Whenever either the primary or the backup executions finishes, mark it as completed
  - The problem of "stragglers" (slow workers)
    - Other jobs consuming resources on machine
    - Bad disks with soft errors transfer data very slowly
    - Weird things: processor caches disabled (!!)

Fault Tolerance

- Input error: bad records
  - Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix, but not always possible
  - On segment fault
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - Skip bad records
    - If master sees two failures for same record, next worker is told to skip the record

MapReduce Implementations



Twister for Iterative MapReduce

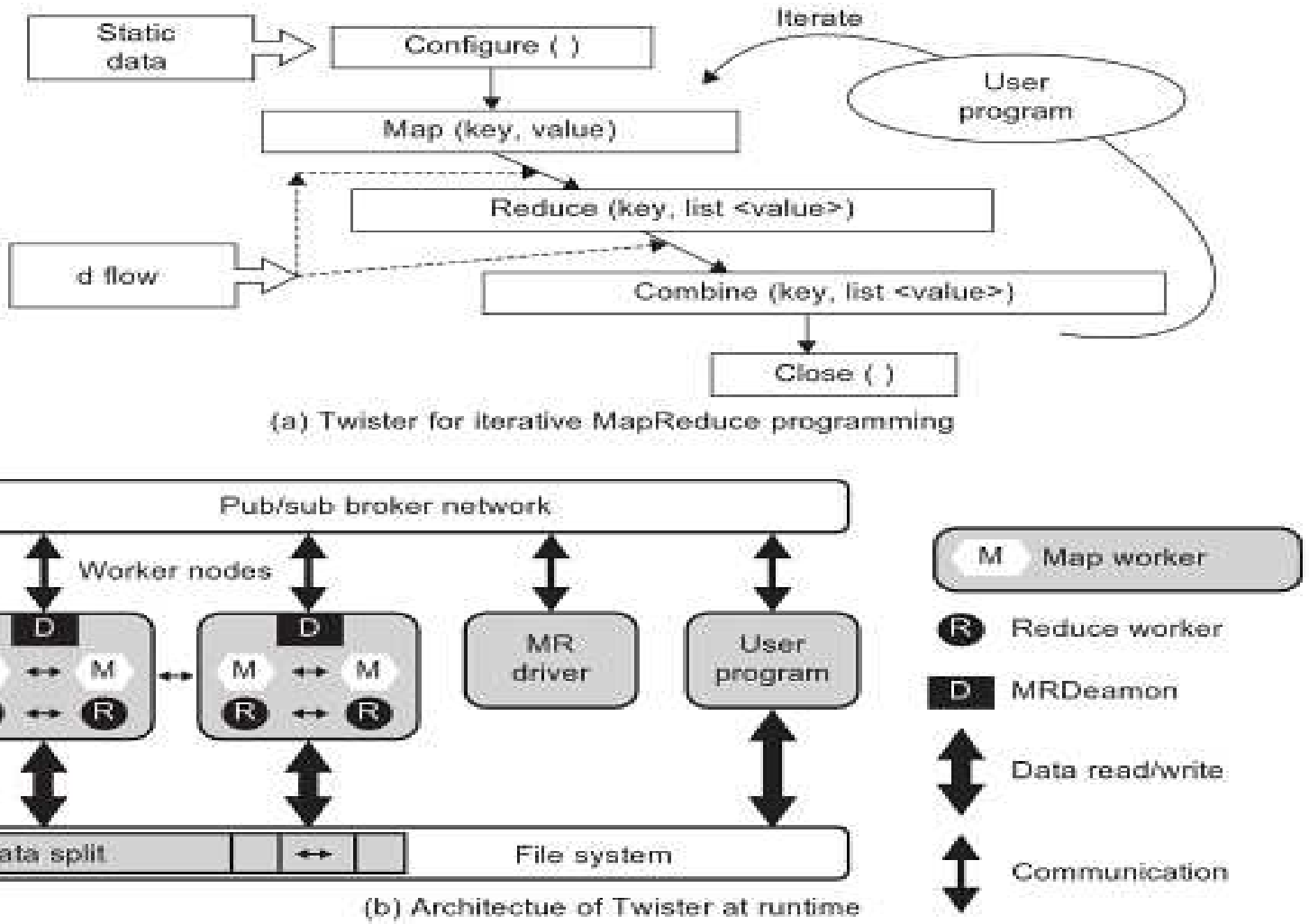
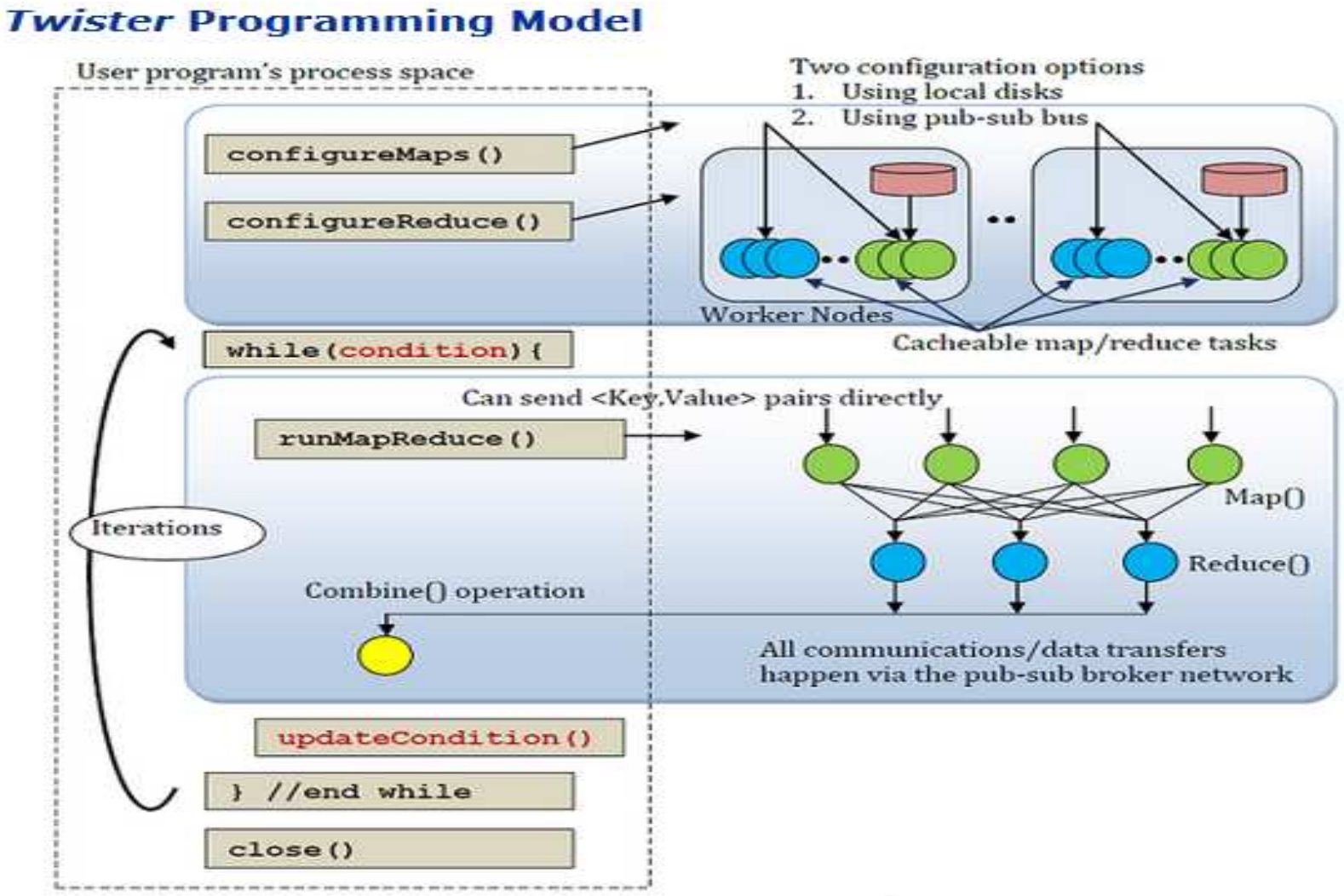


FIGURE 6.7

Twister: An iterative MapReduce programming paradigm for repeated MapReduce executions.

Twister for Iterative MapReduce



Iterative MapReduce programming model using Twister

Performance of Parallel Programming Models

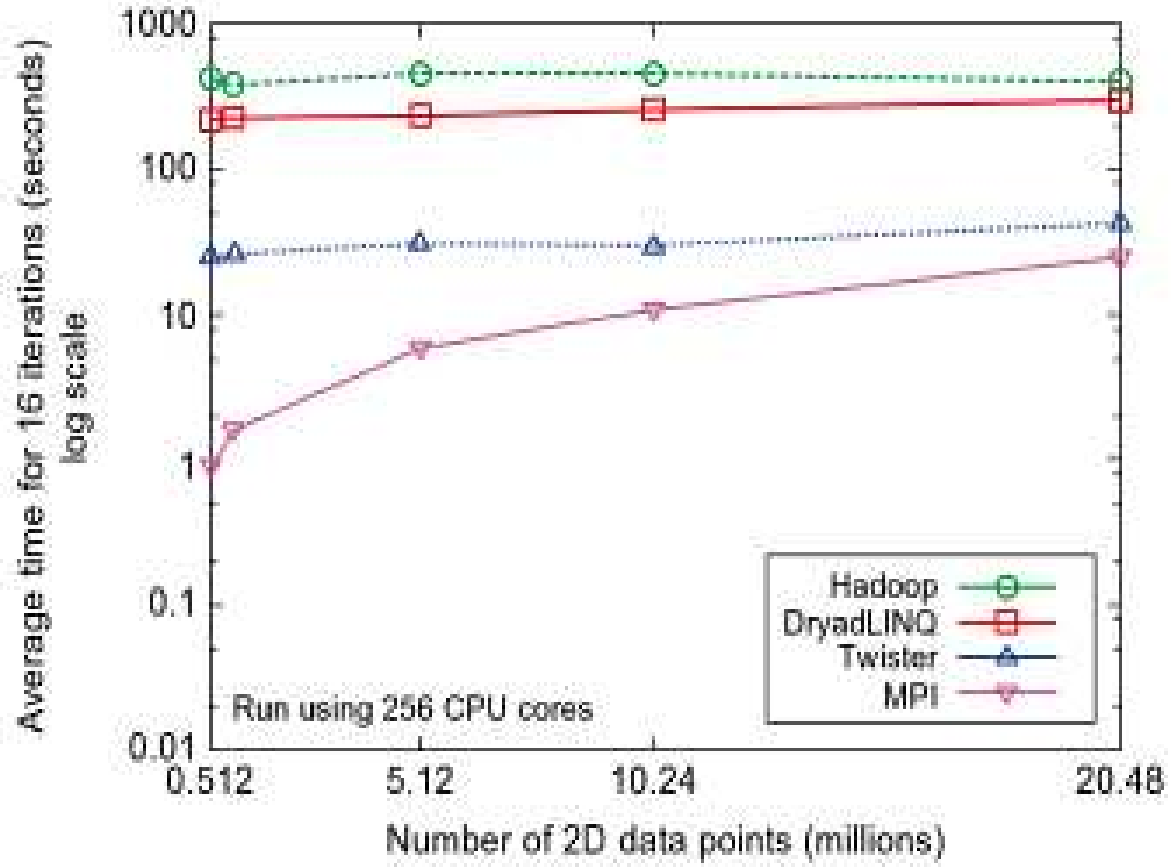


FIGURE 6.8

Performance of K means clustering for MPI, Twister, Hadoop, and DryadLINQ.

Hadoop MapReduce

- Hadoop is an open source implementation of MapReduce coded and released in Java by Apache.
- Hadoop implementation uses Hadoop Distributed File System.
- Hadoop core is divided into two layers.
  - **MapReduce Engine** as computation engine
  - **HDFS** as data storage manager.
  - HDFS is inspired by GFS, it organizes files and stores data in distributed computing system.

Hadoop MapReduce

Hadoop : software platform originally developed by Yahoo enabling users to write and run applications over vast distributed data.

Attractive Features in Hadoop :

- Scalable : can easily scale to store and process petabytes of data in the Web space
- Economical : An open-source MapReduce minimizes the overheads in task spawning and massive data communication.
- Efficient: Processing data with high-degree of parallelism across a large number of commodity nodes
- Reliable : Automatically maintains multiple copies of data to facilitate redeployment of computing tasks on failures

Basic Features of a File System

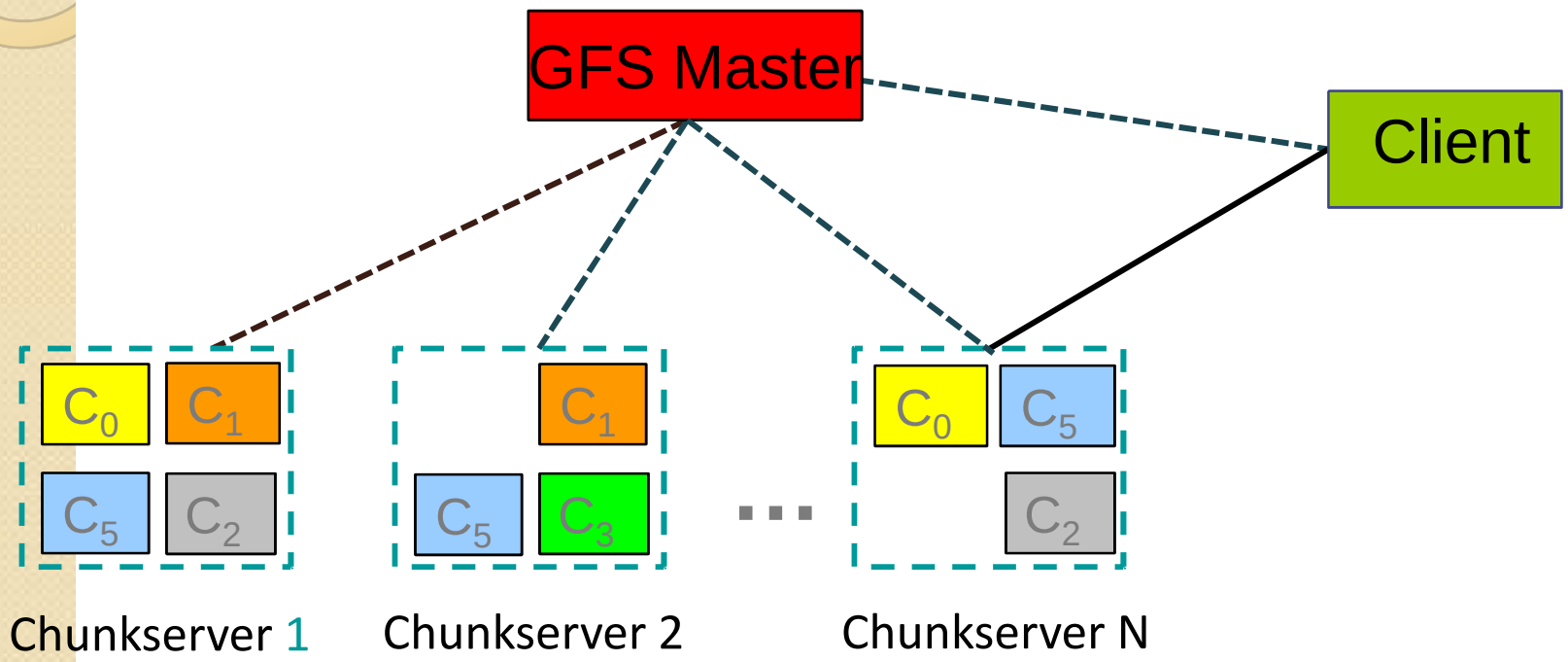
- A Distributed File system must provide the following features
  - Performance
  - Scalability
  - Concurrency control
  - Fault Tolerance
  - Security requirements

Basic Features: HDFS

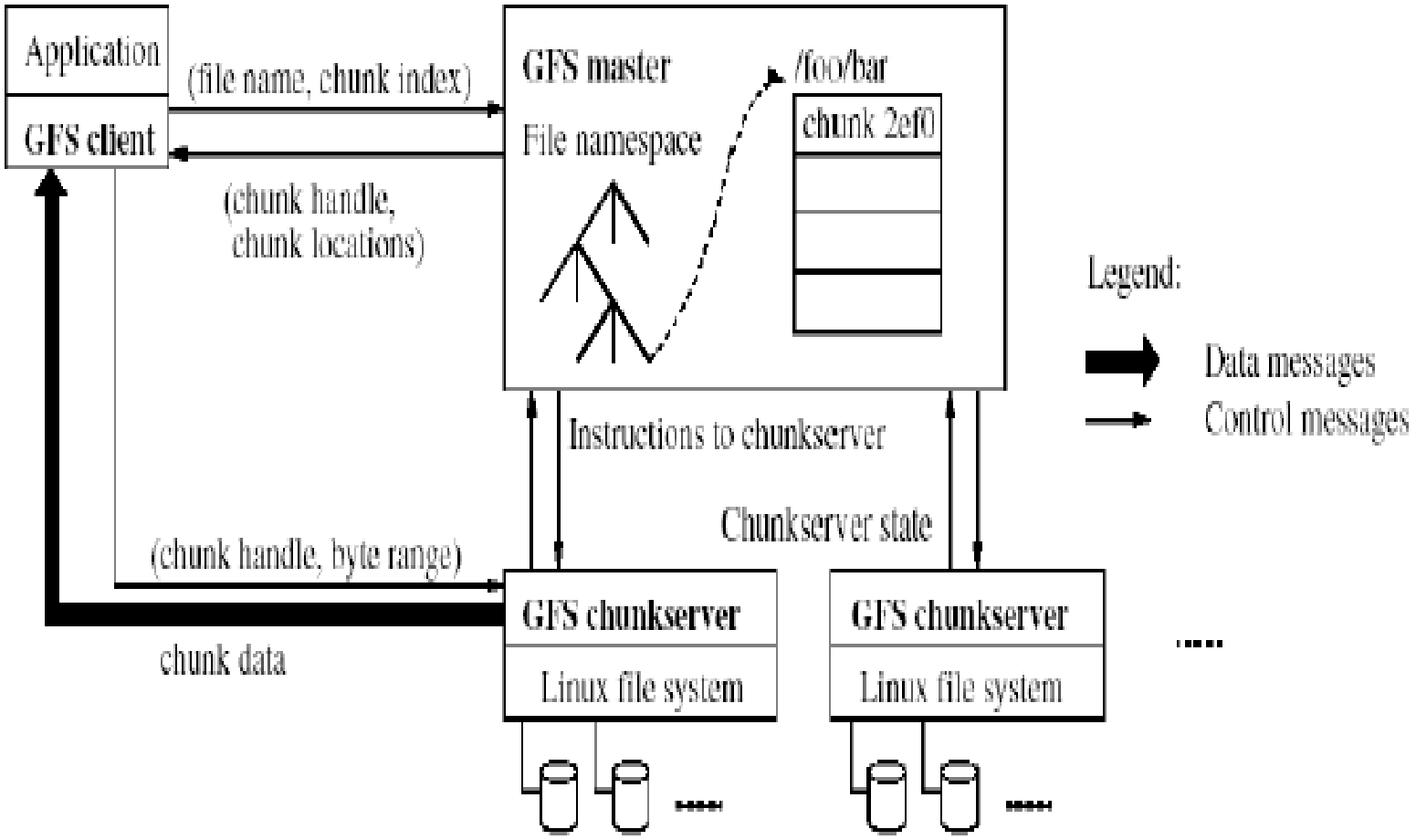
- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware
- Security is not supported by HDFS

ARCHITECTURE OF GFS

GFS architecture



GFS architecture

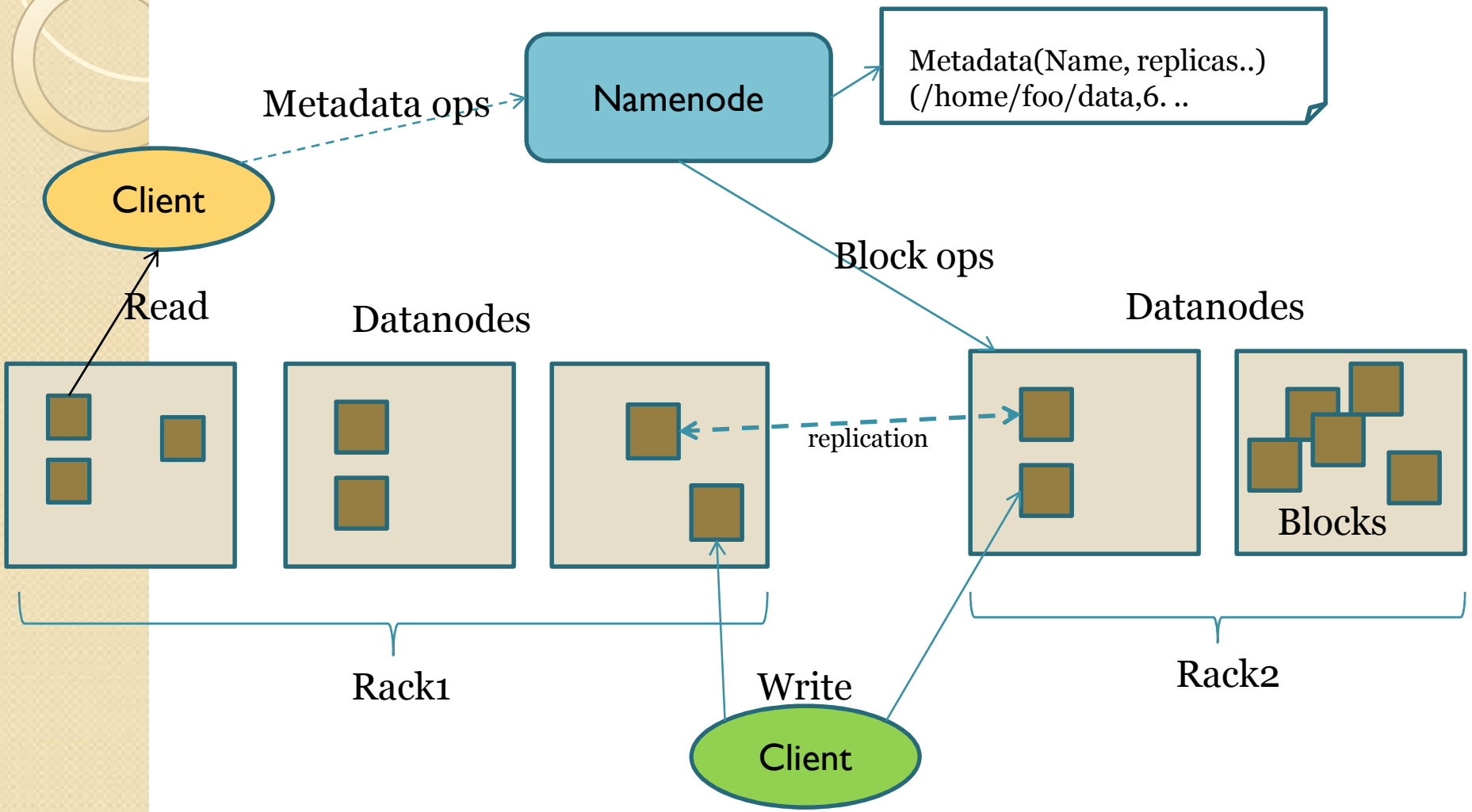


ARCHITECTURE OF HDFS

Namenode and Datanodes

- Master/slave architecture
- HDFS cluster consists of a single **Namenode**, a master server that manages the file system namespace, Metadata and regulates access to files by clients.
- There are a number of **DataNodes** usually one per node in a cluster.
- The DataNodes manage storage attached to the nodes that they run on.
- HDFS exposes a file system namespace and allows user data to be stored in files.
- A file is split into one or more blocks and set of blocks are stored in DataNodes.
- DataNodes: serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode.

HDFS Architecture



File system Namespace

- **Hierarchical file system** with directories and files
- Create, remove, move, rename etc.
- **Namenode** maintains the **file system**
- Any **meta information** changes to the file system recorded by the **Namenode**.
- An application can specify the number of **replicas** of the file needed: replication factor of the file. This **information** is stored in the **Namenode**.

Fault tolerance

- Failure is the norm rather than exception
- Hadoop is designed to be deployed on **low-cost** hardware by default, a **hardware failure** in this system is considered to be **common** rather than an exception.
- Hadoop considers the following issues to fulfill **reliability** requirements of the file system
- **Block replication**
- **Replica placement**
- **Heartbeat and Blockreport messages**

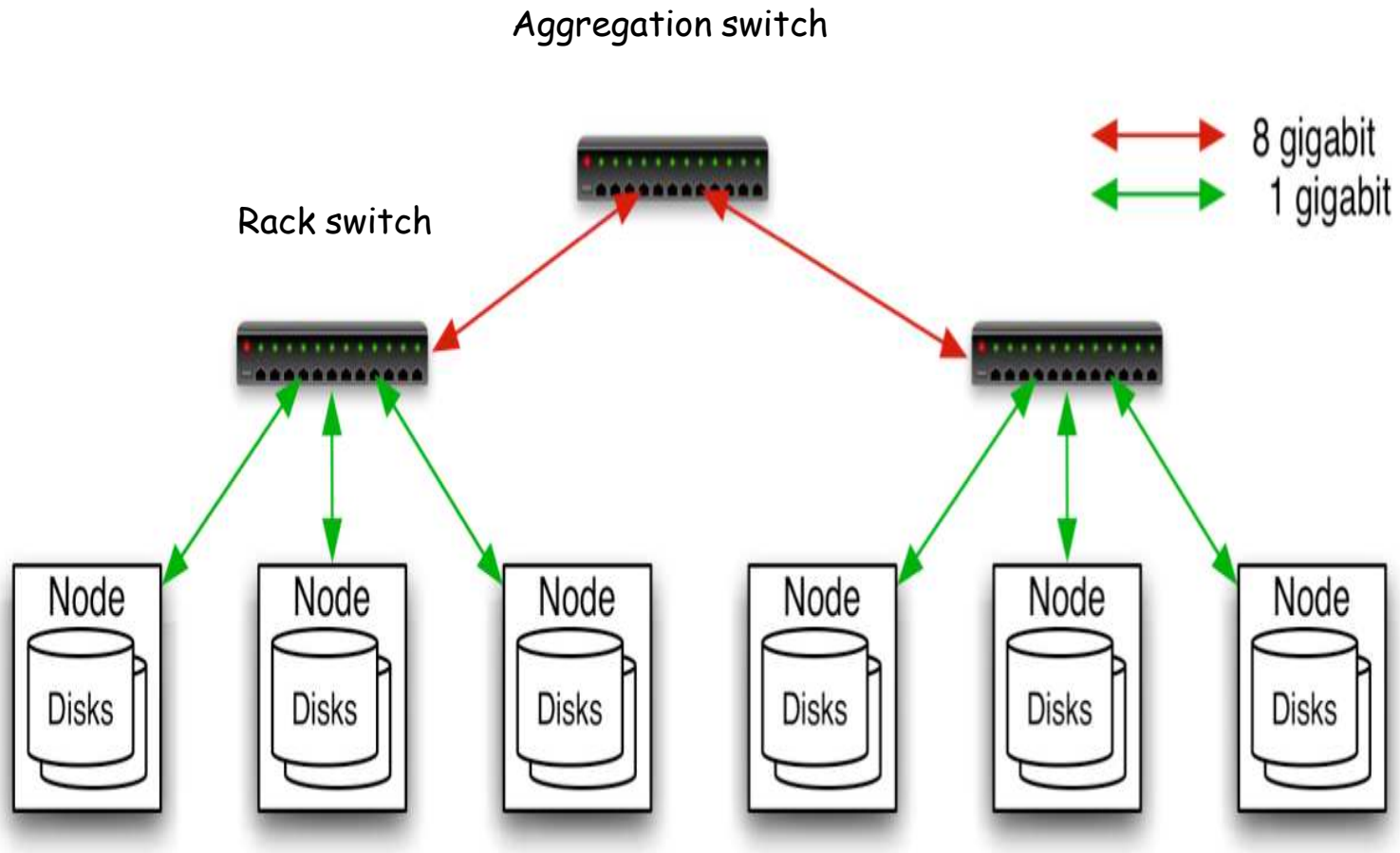
Fault tolerance

- **Block replication**
  - HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster.
  - The replication factor is set by the user and is three by default.
- **Replica placement**
  - Although storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability, it is sometimes ignored as the cost of communication between two nodes in different racks is relatively high
  - Sometimes HDFS compromises its reliability to achieve lower communication costs.
  - Stores one replica in the same node the original data is stored, one replica on a different node but in the same rack, and one replica on a different node in a different rack to provide three copies of the data

Replica Placement

- The placement of the replicas is critical to HDFS reliability and performance.
- Optimizing replica placement distinguishes HDFS from other distributed file systems.
- **Rack-aware replica placement:**
  - Goal: improve reliability, availability and network bandwidth utilization
  - Research topic
- **Many racks, communication between racks are through switches.**
- **Network bandwidth between machines on the same rack is greater than those in different racks.**
- **Namenode determines the rack id for each DataNode.**
- Replicas are typically placed on unique racks
  - Simple but non-optimal
  - Writes are expensive
  - Replication factor is 3
  - Another research topic?
- **Replicas are placed: one on a node in a local rack, one on a different node in the local rack and one on a node in a different rack.**
- 1/3 of the replica on a node, 2/3 on a rack and 1/3 distributed evenly across remaining racks.

Replica Placement



Replica Selection

- Replica selection for READ operation: HDFS tries to **minimize** the **bandwidth consumption and latency**.
- If there is a **replica** on the **Reader node** then that is preferred.
- HDFS cluster may span **multiple data centers**: replica in the **local data center** is **preferred** over the **remote** one.

Fault tolerance

- **Heartbeat and Blockreport messages**
  - Heartbeats and Block reports are periodic messages sent to the **NameNode** by each **DataNode** in a cluster for every **3 seconds**.
  - Receipt of a **Heartbeat** implies that the **DataNode** is **functioning properly**, while each **Blockreport** contains a **list** of **all blocks** on a **DataNode**.
  - The NameNode receives such messages because it is the sole **decision maker** of **all replicas** in the system.

HDFS High-Throughput Access to Large Data Sets (Files)

- HDFS is primarily designed for **batch processing** rather than **interactive processing** data access **throughput** in HDFS is more important than **latency**
- Individual files are broken into **large blocks** (e.g., **64 MB**) to allow HDFS to **decrease** the **amount** of **metadata storage** required per file.
- 2 Advantages
  - **List** of **blocks** per file will **shrink** as the size of individual **blocks increases**
  - Keeping large amounts of **data sequentially** within a block, HDFS provides **fast streaming reads** of data.

Streaming Data

- Streaming access enables data to be transferred in the form of a steady and continuous stream.
- This means if data from a file in HDFS needs to be processed, HDFS starts **sending** the **data** as it **reads** the file and **does not wait** for the entire file to be read.
- The client who is consuming this data starts **processing** the data **immediately**, as it receives the stream from HDFS.
- This makes data **processing** really **fast.**
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency. No Data Appending as in GFS

HDFS Operations

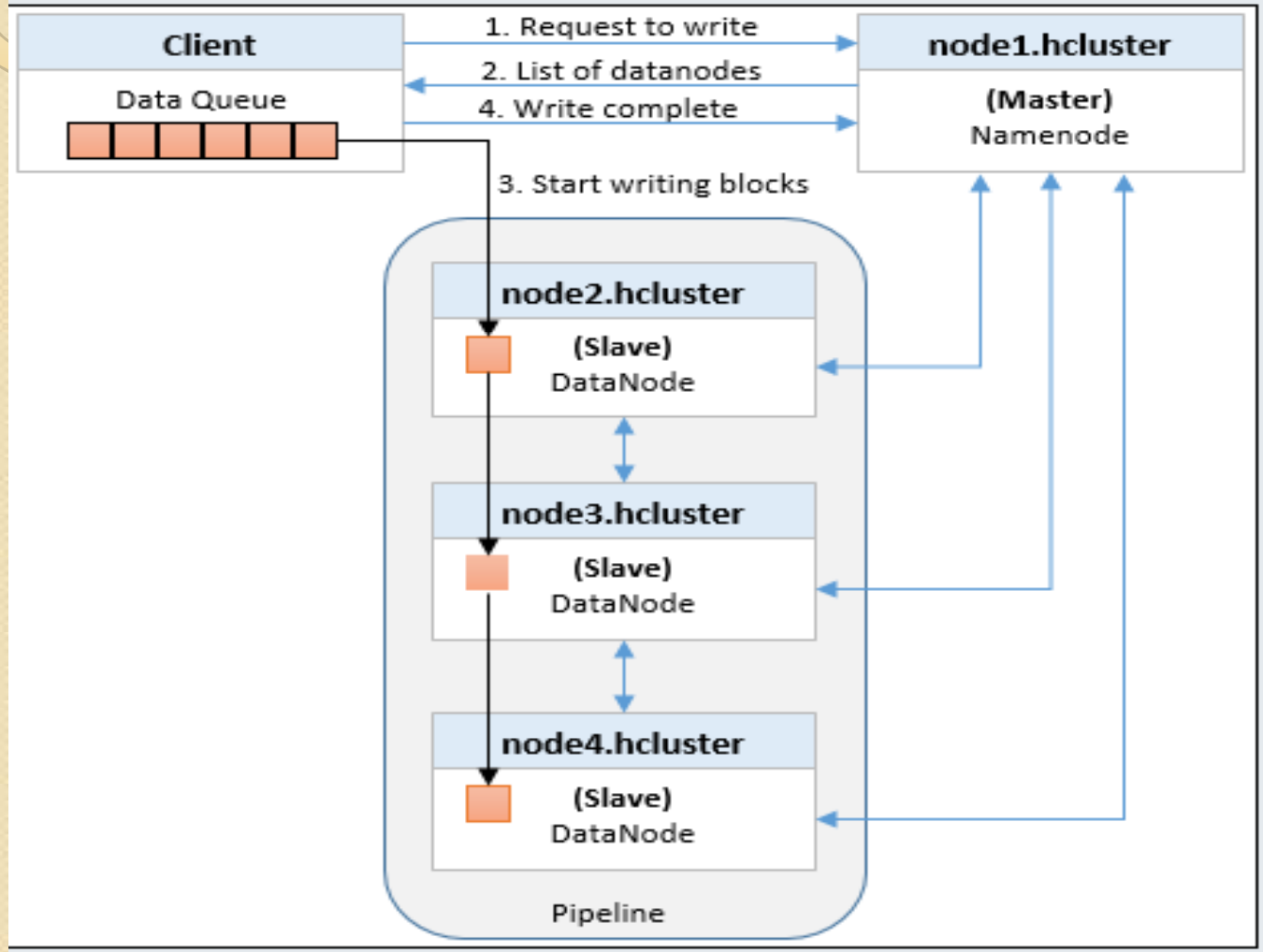
Writing a file into HDFS

Writing files in HDFS

- To write a file in HDFS, a user sends a "create" request to the NameNode to create a new file in the file system namespace.
- If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the write function.
- The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode.
- Since each file block needs to be replicated by a predefined factor, the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.
- The steamer then stores the block in the first allocated DataNode.
- Afterward, the block is forwarded to the second DataNode by the first DataNode.
- The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode.
- Once this replication process is finalized, the same process starts for the second block and continues until all blocks of the file are stored and replicated on the file system.

Writing Data into HDFS

The following diagram shows the data block replication process across the datanodes during a write operation in HDFS:



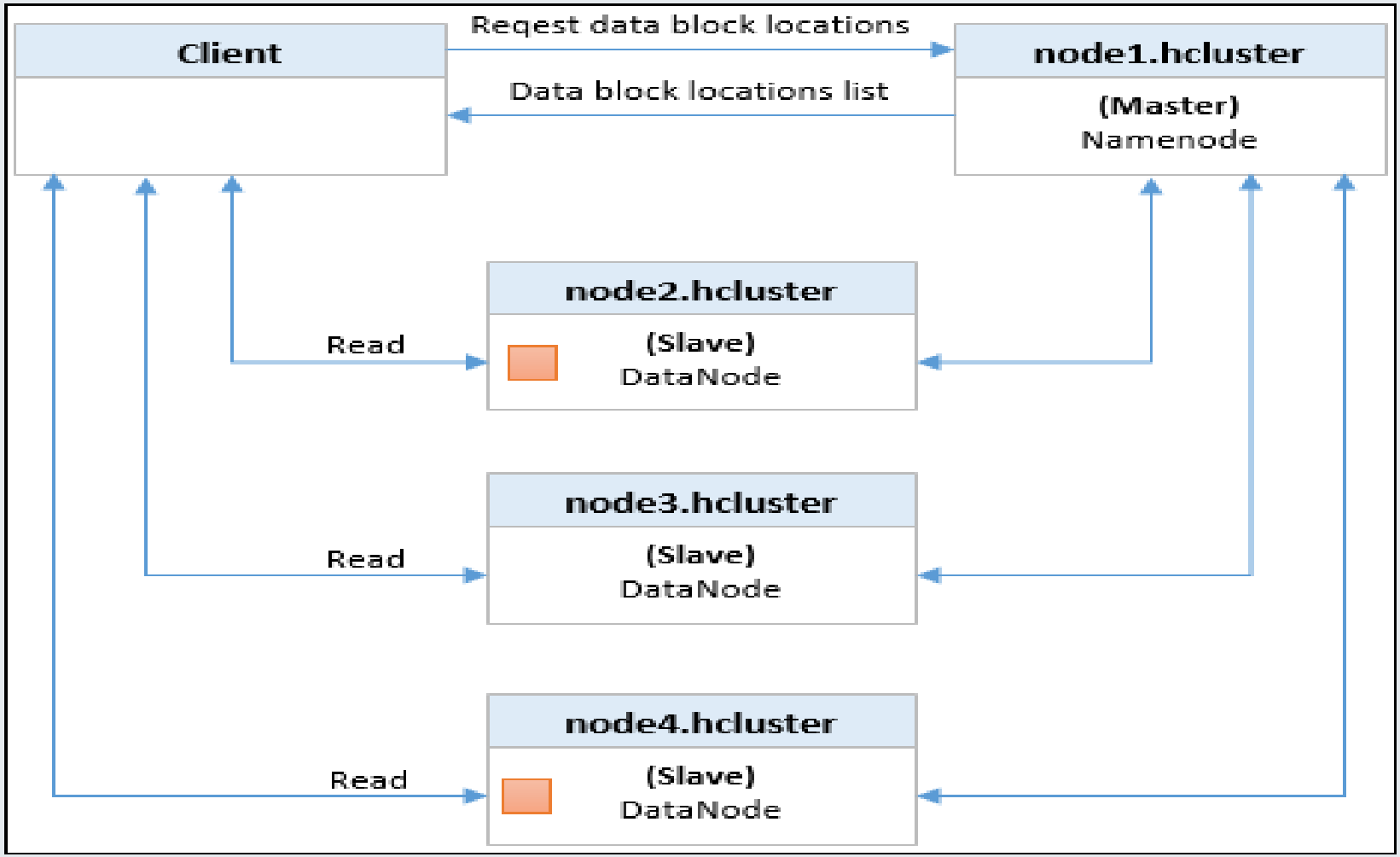
Reading a file from HDFS

Reading files from HDFS

- To read a file in HDFS, a user sends an "open" request to the NameNode to get the location of file blocks.
- For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file.
- The number of addresses depends on the number of block replicas.
- Upon receiving such information, the user calls the *read function to connect to the* closest DataNode containing the first block of the file.
- After the first block is streamed from the respective DataNode to the user, the established connection is terminated
- The same process is repeated for all blocks of the requested file until the whole file is streamed to the user.

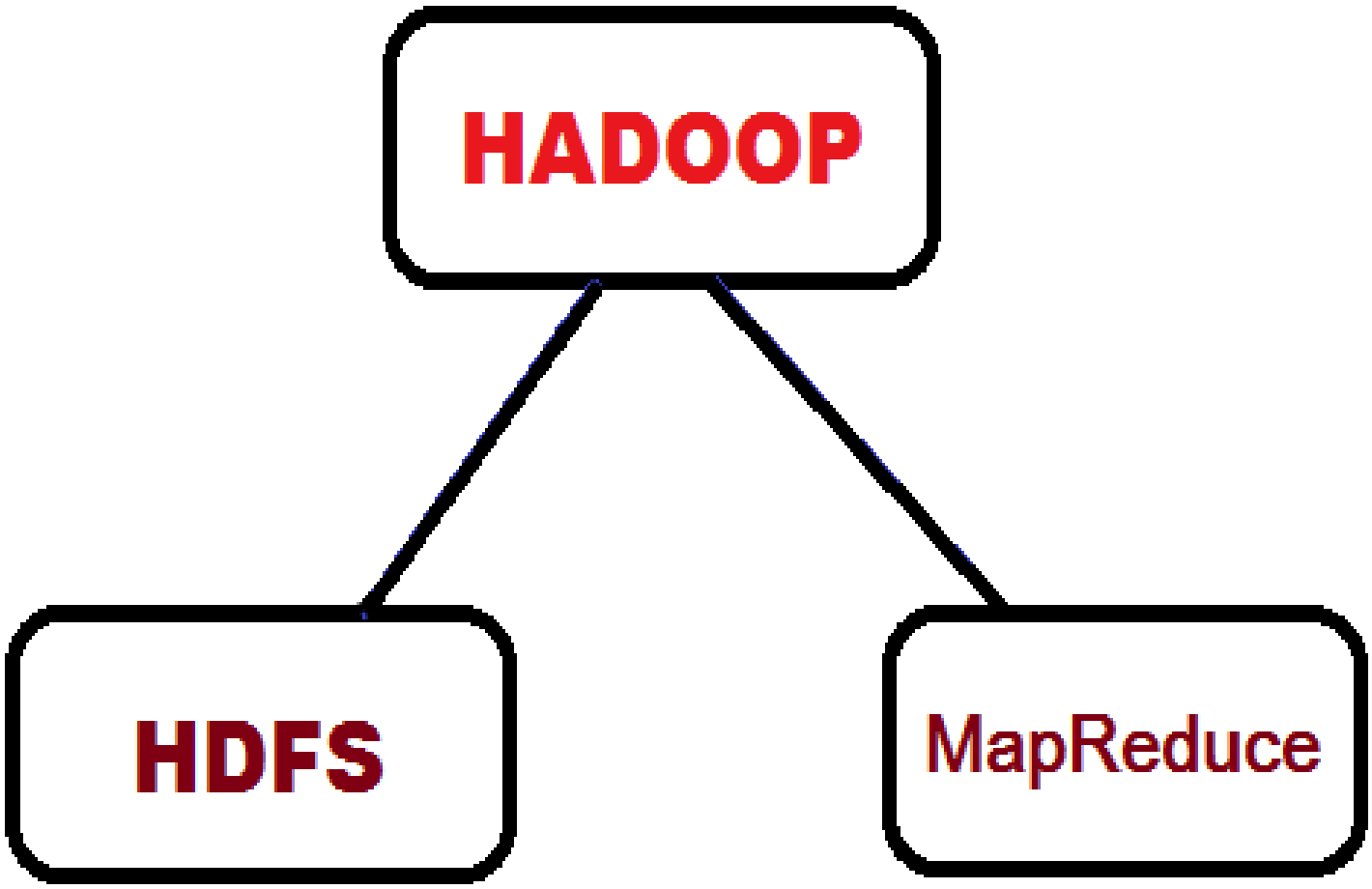
Reading Data from HDFS

The following diagram shows the read operation of a file in HDFS:

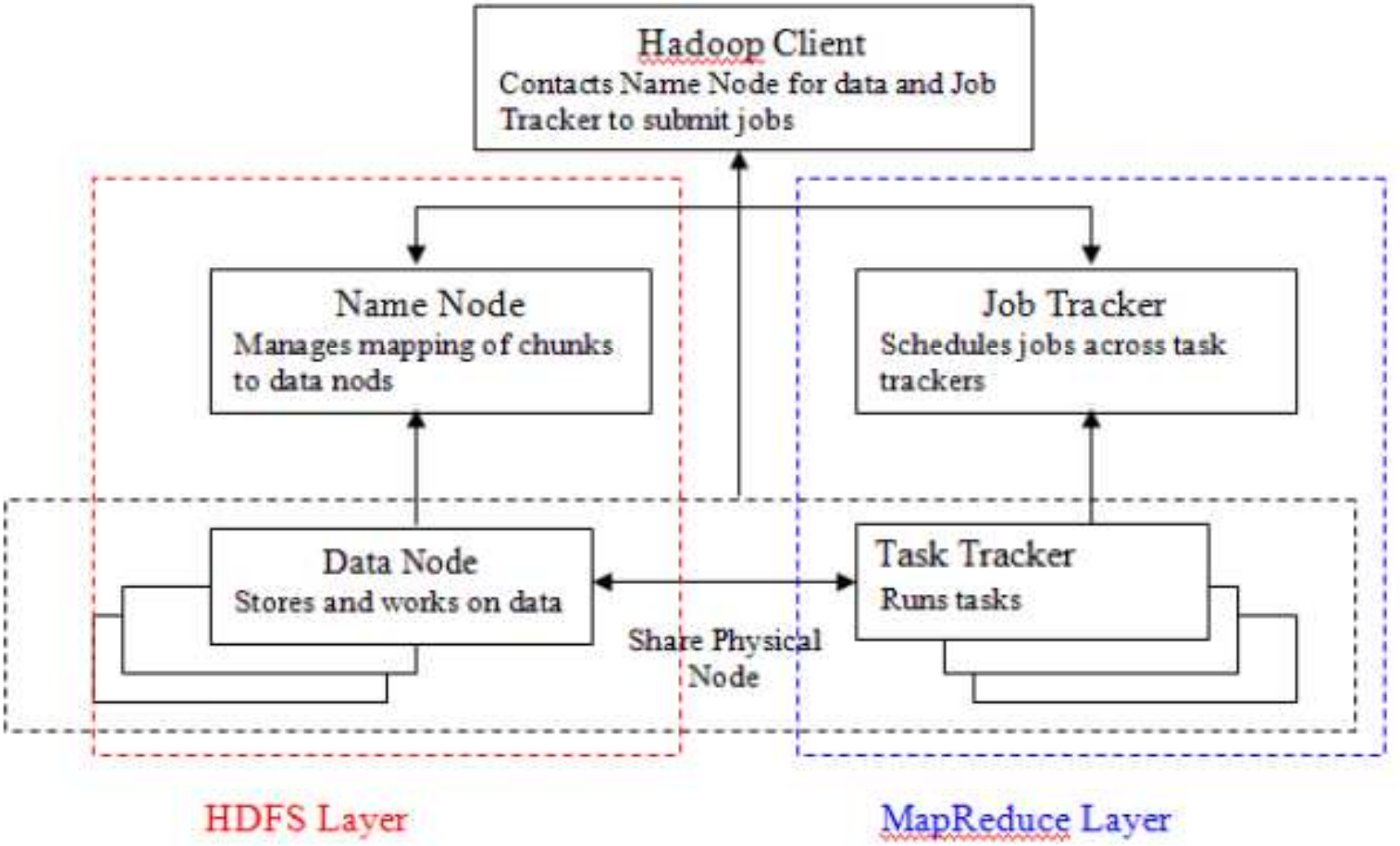


ARCHITECTURE OF HADOOP

Hadoop Architecture



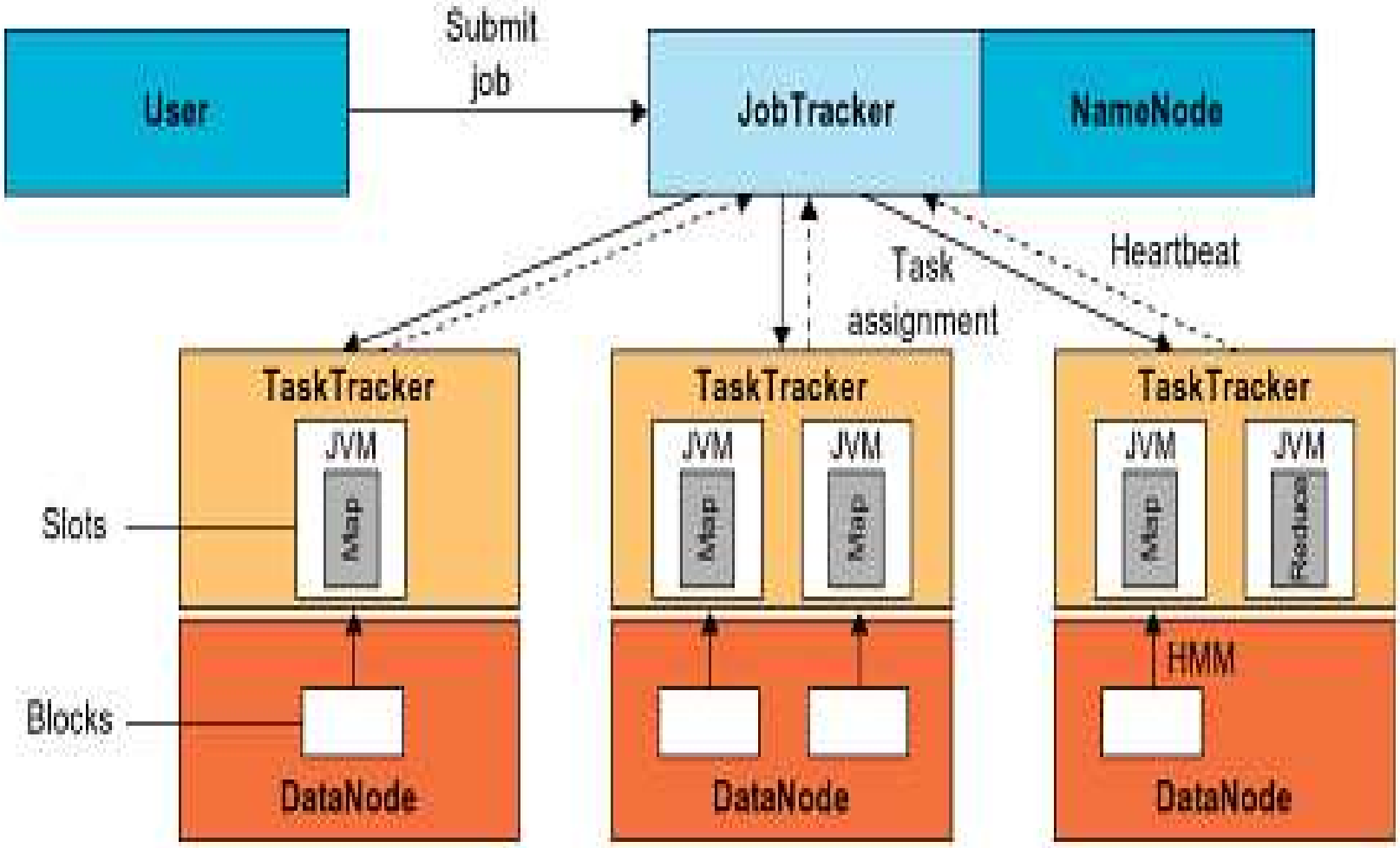
Hadoop Architecture



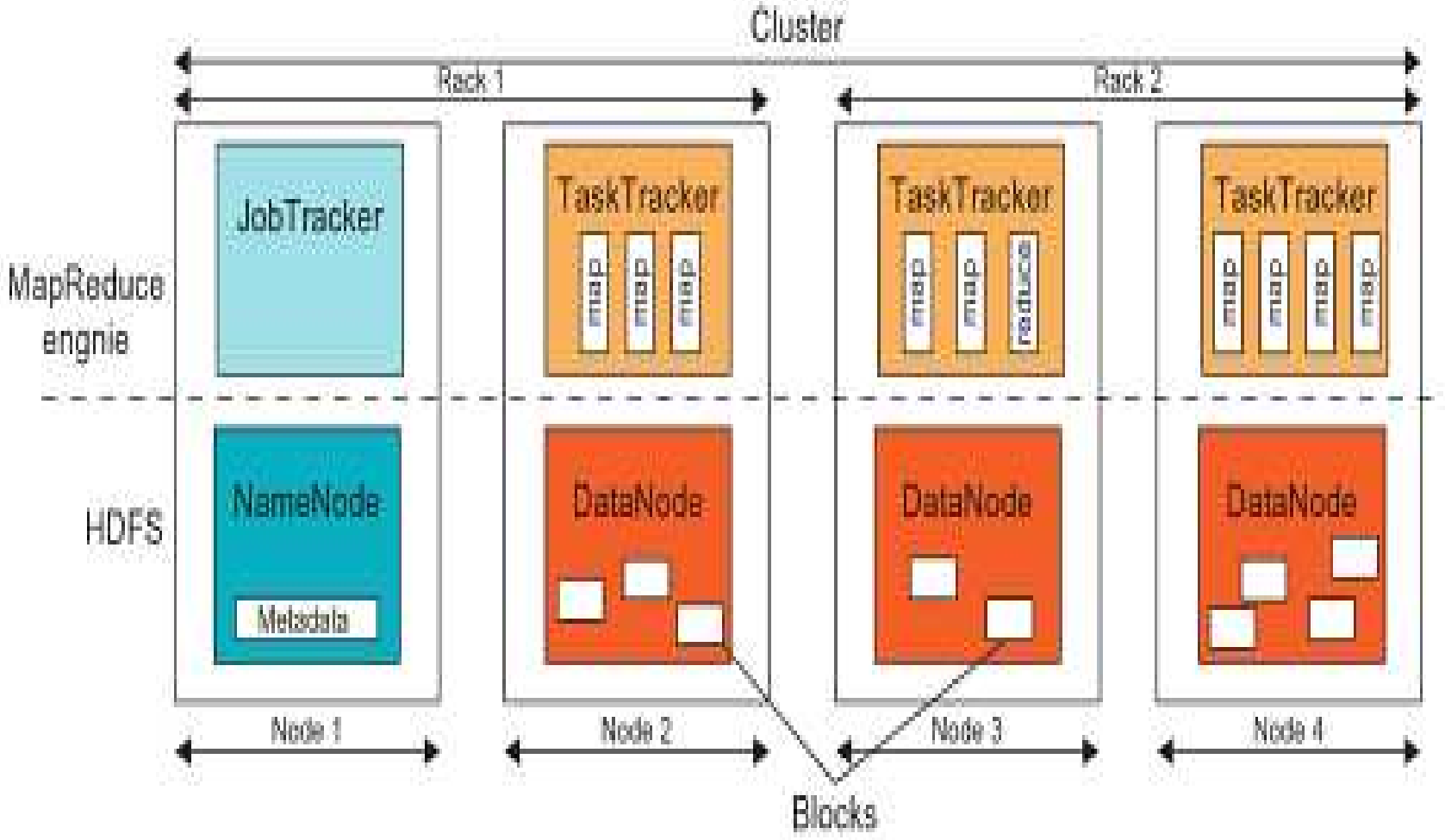
Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- MapReduce framework
  - Executes user jobs specified as "map" and "reduce" functions
  - Manages work distribution & fault-tolerance

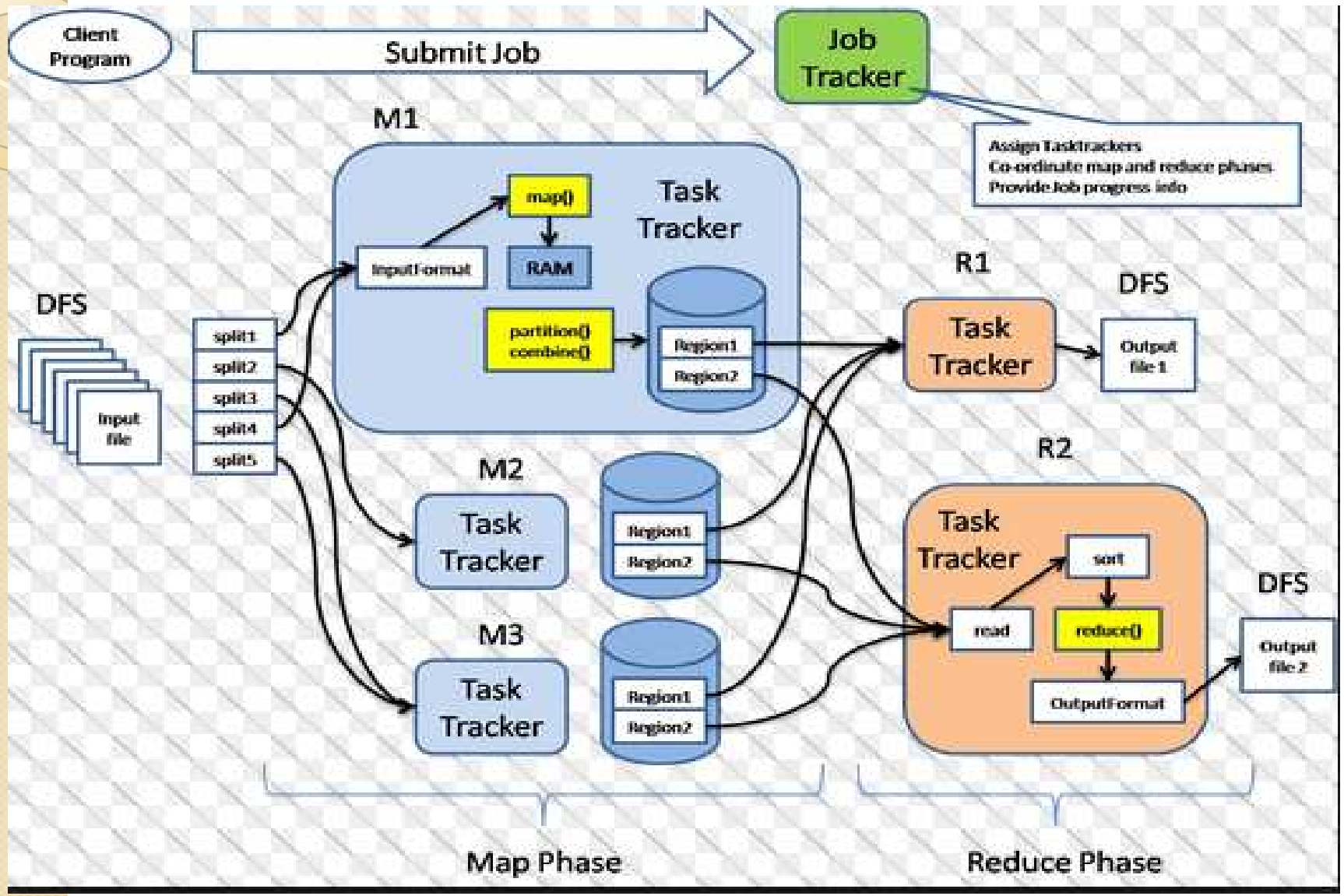
Hadoop Architecture



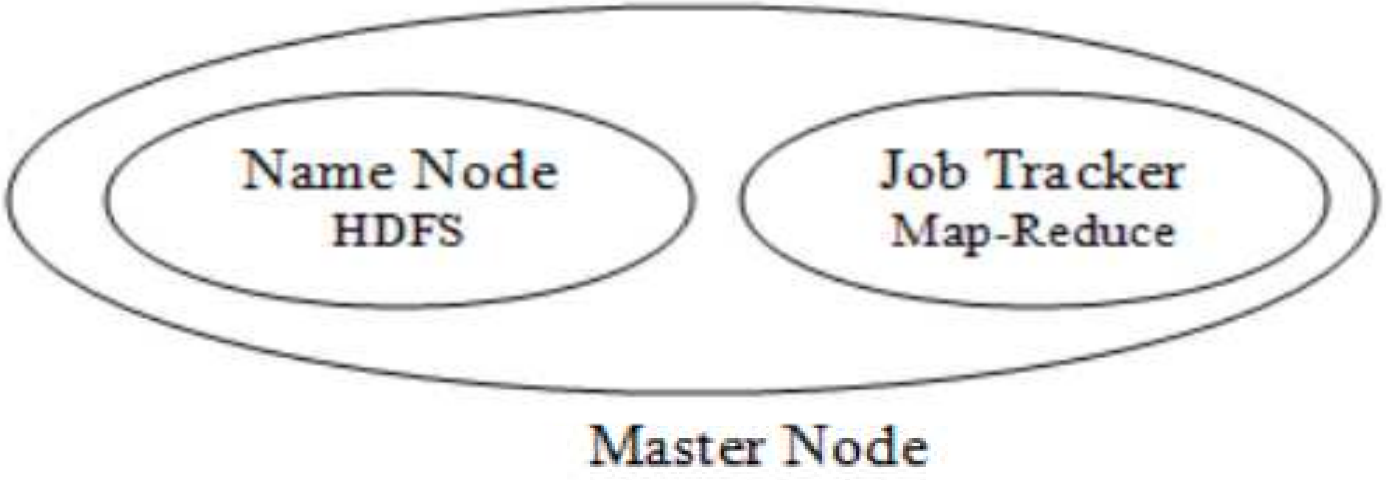
HDFS and MapReduce in Hadoop



Hadoop Architecture



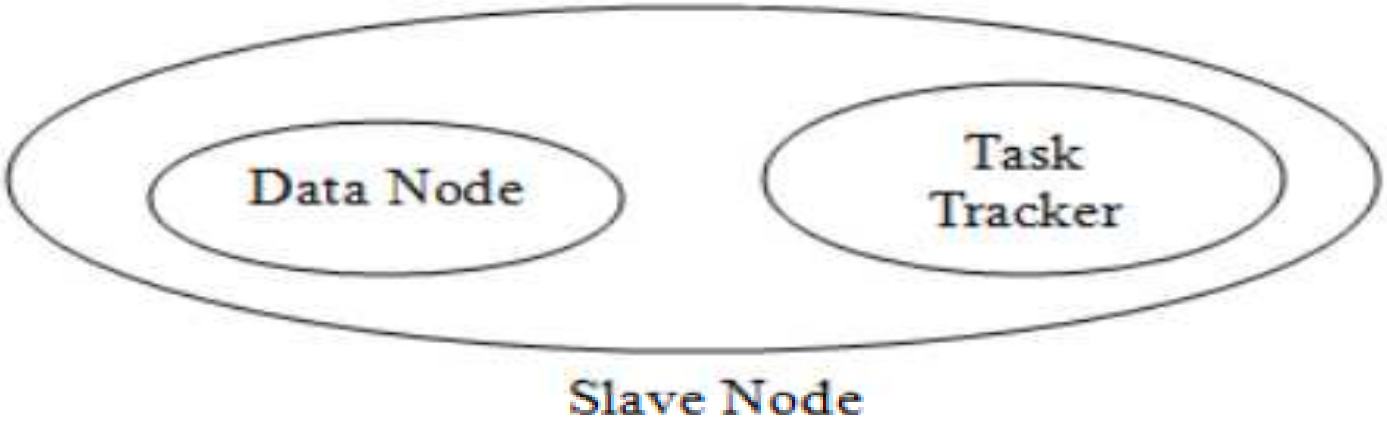
Hadoop Architecture



These functions are handled by two different nodes:

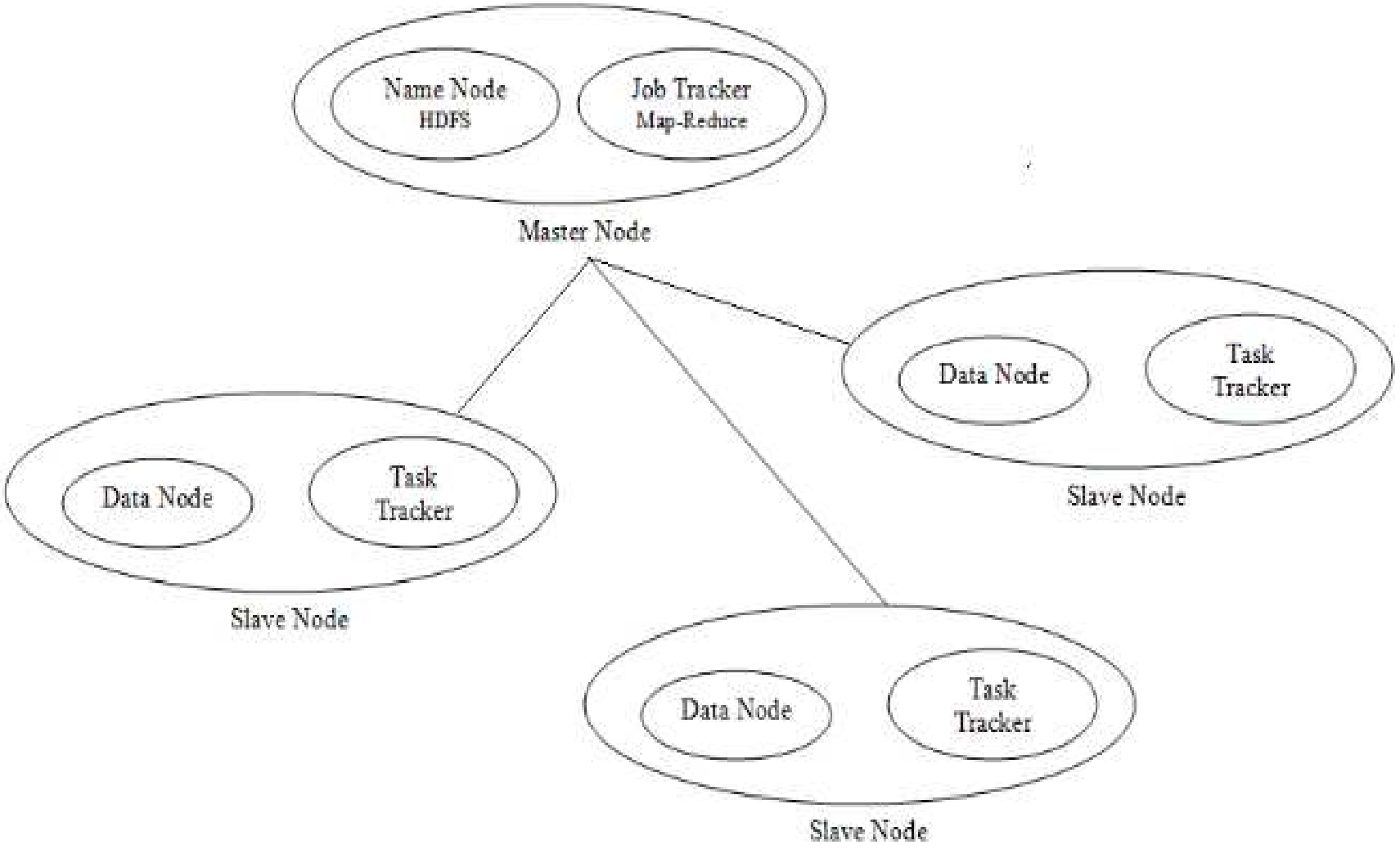
- **Name Node**: helps in coordinating HDFS
- **Job Tracker**: helps in coordinating parallel execution of Map-Reduce

Hadoop Architecture

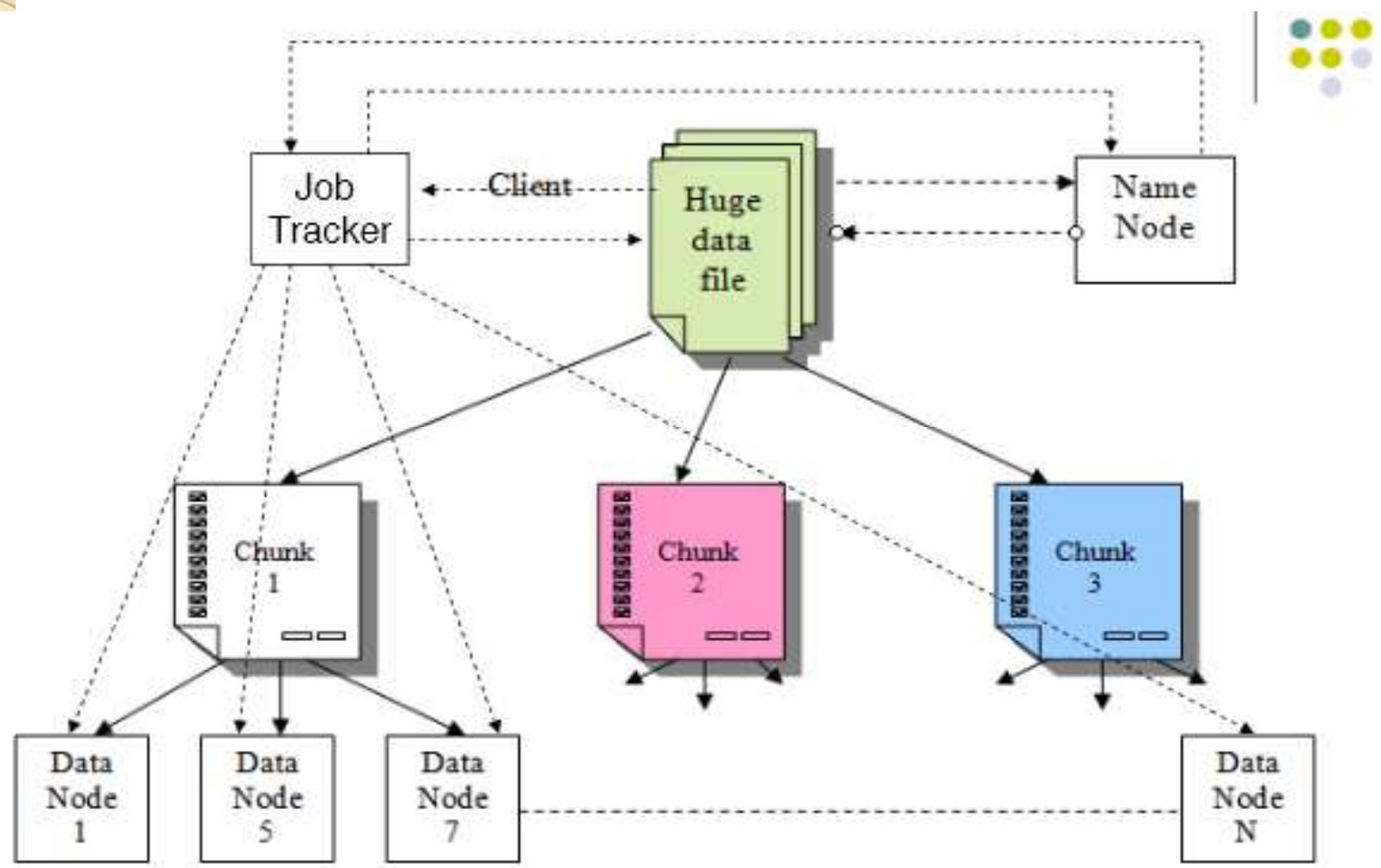


- Slave node consists of Data Node and Task Tracker.
- There are number slave nodes in a Hadoop cluster.
- These nodes communicate and receive instructions for the Master nodes and perform the assigned tasks.

Hadoop Architecture



Hadoop Working



Name Node and Data Node

- **Name Node** is a critical component of the HDFS.
- Responsible for the distribution of the data throughout the Hadoop cluster.
- Obtains the data from client machine divides it in to chunks.
- Distributes same data chunk to three different data nodes leading to data redundancy
- Keeps track of What chucks belong to a file and which Data Node holds its copy.
- Making sure each chunk of file has the minimum number of copies in the cluster as required.
- Directs clients for write or read operation
- Schedule and execute Map Reduce jobs
- One slave act as secondary NameNode

Data Node

- Responsible to store the chunk of data that assigned to it by the Name Node.

Job Tracker and Task Tracker

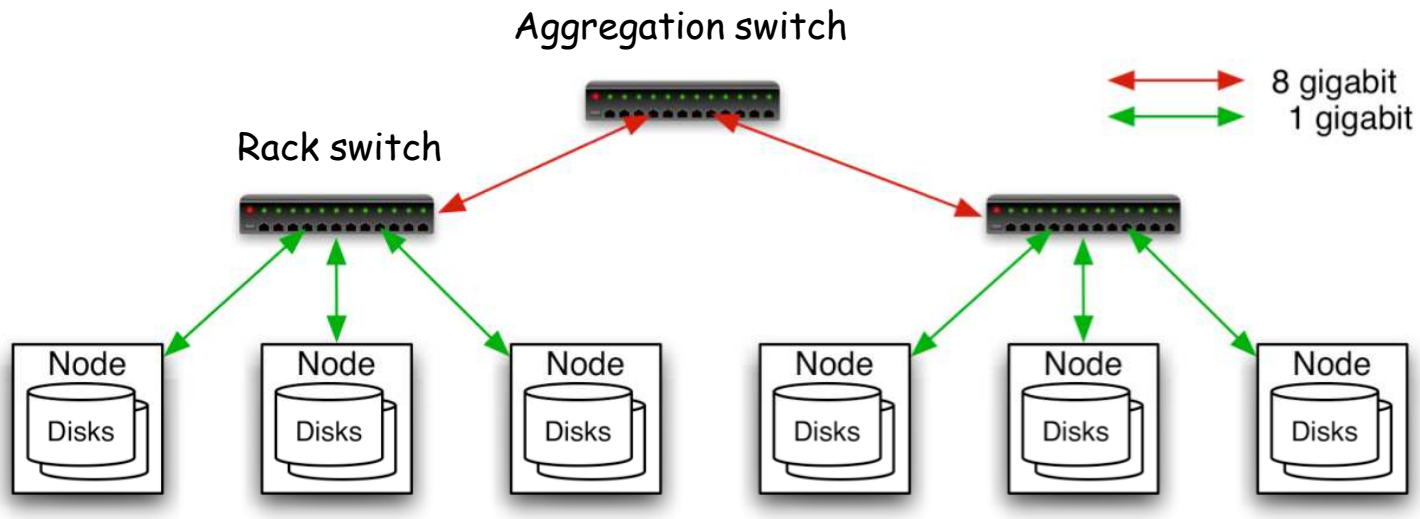
Job Tracker

- Client submitted Map-Reduce logic.
- Responsible for scheduling the task to the slave nodes.
- Job Tracker consults the Name Node and assigns the task to the nodes which has the data on which this task would be performed
- Locality of data is checked for assigning a Map function to a node (Task Tracker), whereas Reduce function does not need locality of data.
- One Task Tracker may act as secondary Job Tracker

Task Tracker

- Has the actual logic to perform the task.
- Performs the task (Map and reduce functions) on the data assigned to it by Master Node.

Typical Hadoop Cluster



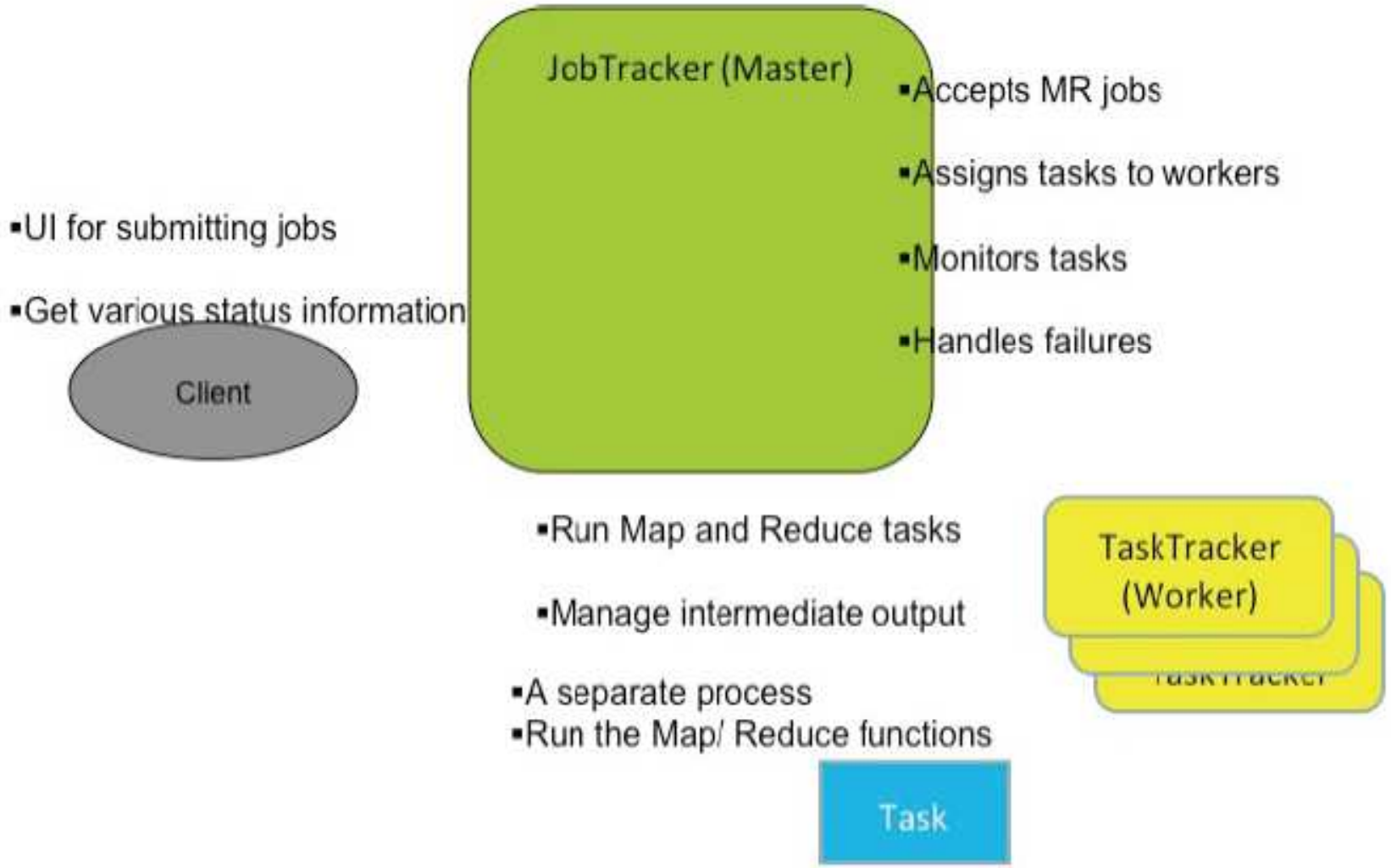
- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort):
  8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf

Typical Hadoop Cluster

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf

MapReduce Architecture



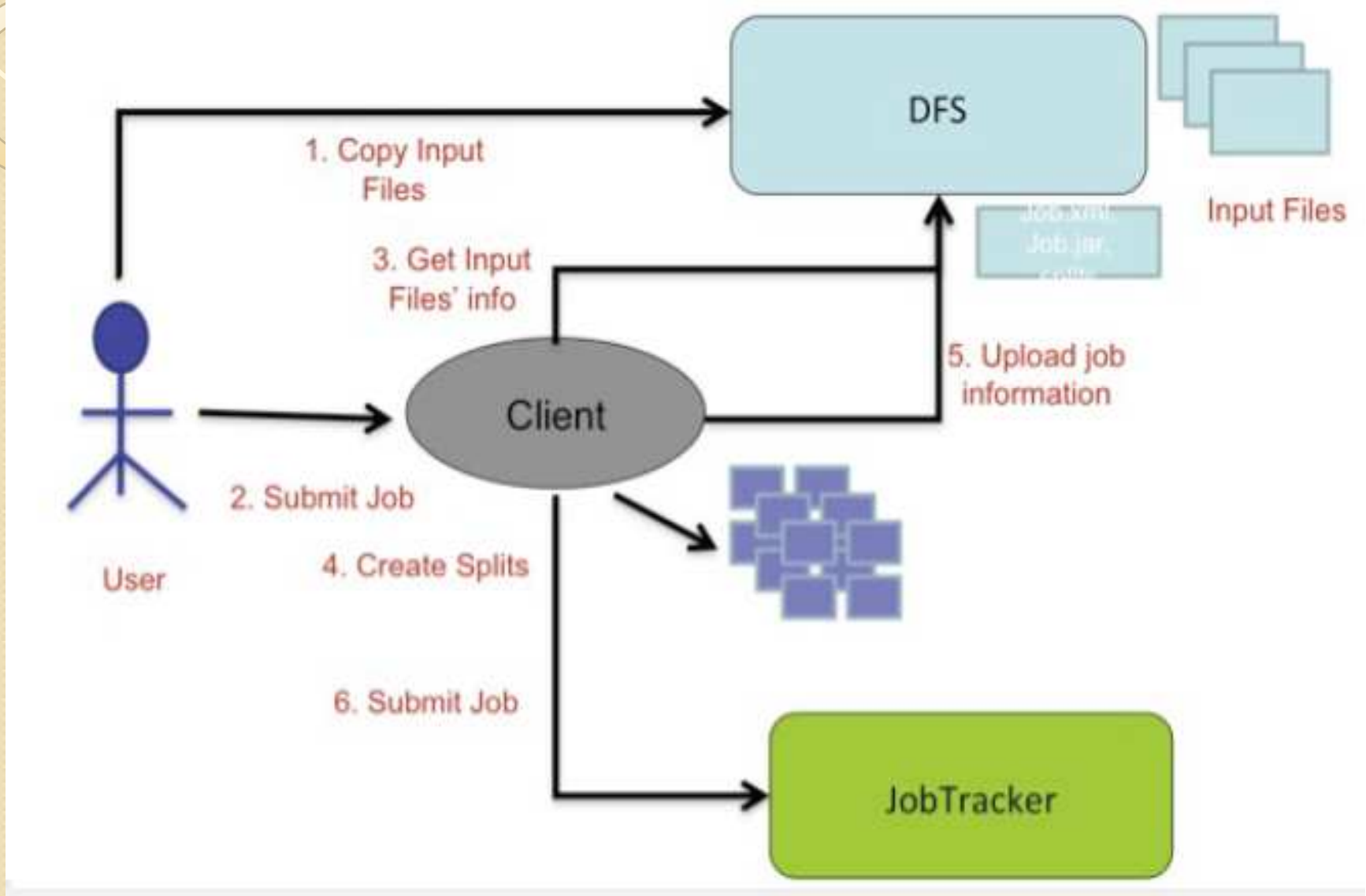
Running a Job in Hadoop

- Job Submission
- Task assignment
- Task execution
- Task running check

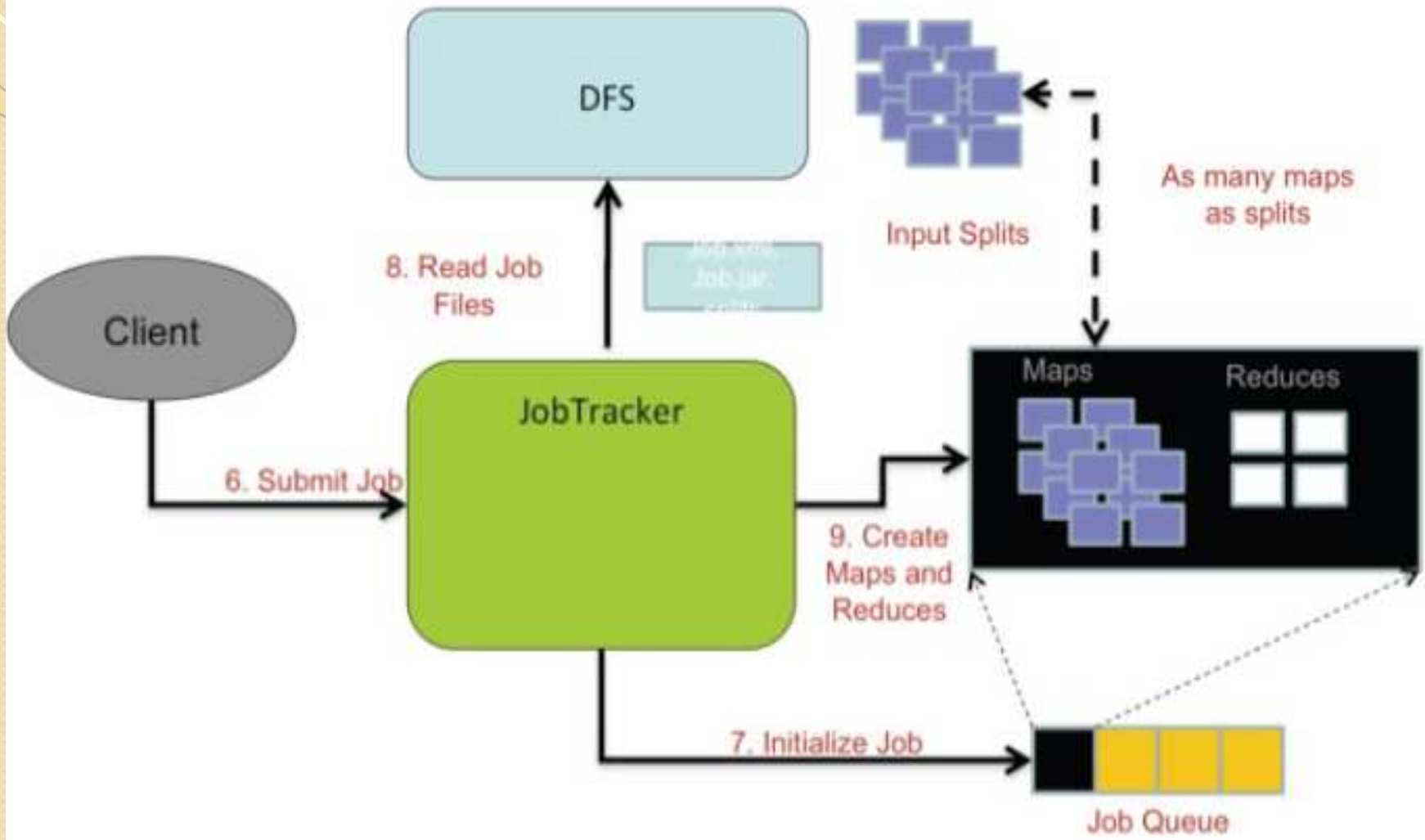
Running a Job in Hadoop

- **Job Submission**
  - Each job is submitted from a **user node** to the **JobTracker** node
  - A user node asks for a **new job ID** from the JobTracker and computes **input file splits**
  - The **user node** copies some resources, such as the job's **JAR** file, **configuration** file, and computed **input splits**, to the **JobTracker's file system**
  - The user node **submits** the job to the **JobTracker** by calling the ***submitJob()*** *function*

Job Submission



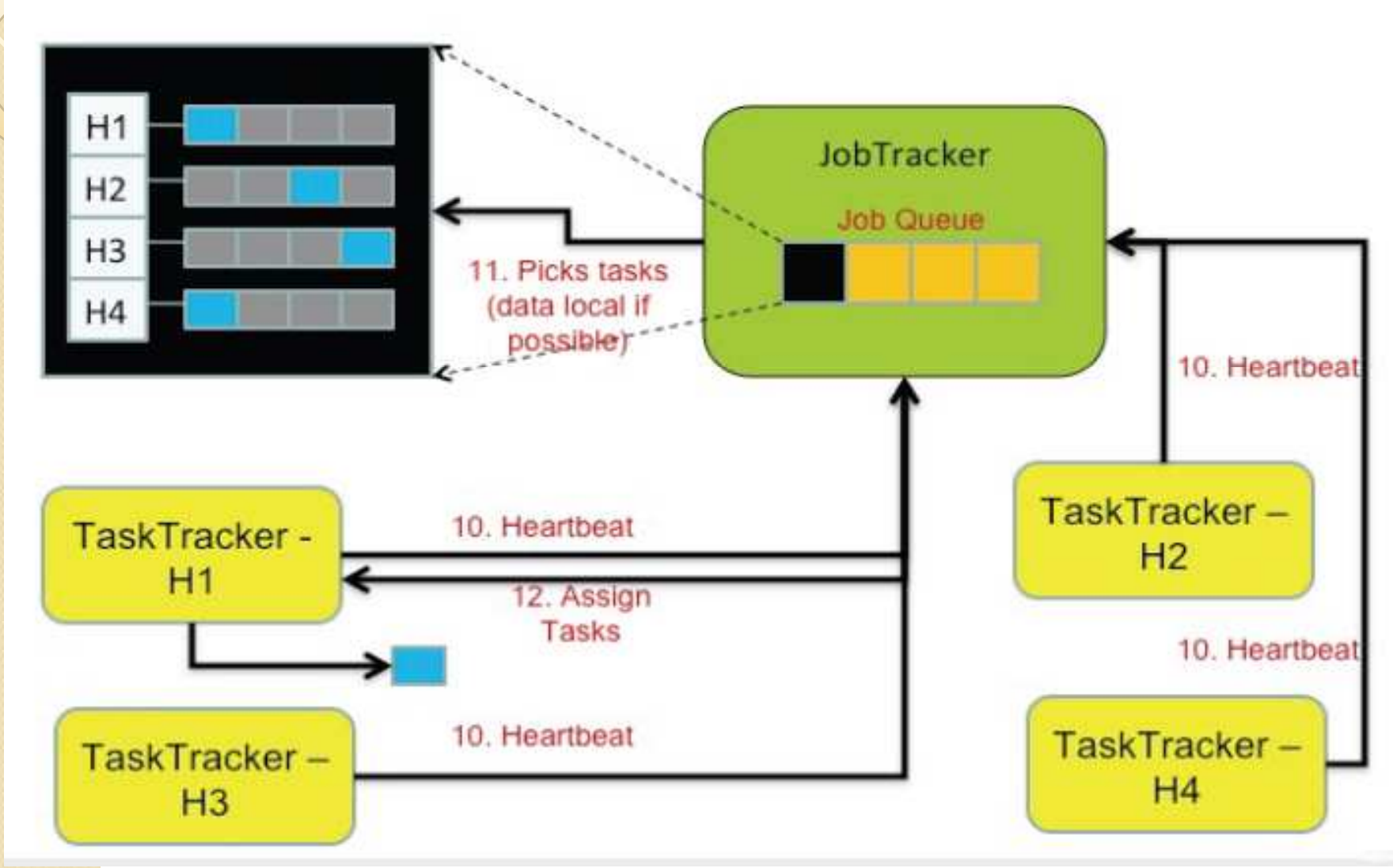
Job Initialization



Running a Job in Hadoop

- **Task Assignment**
  - The **JobTracker** creates **one map task** for **each** computed **input split** by the user node and **assigns** the **map tasks** to the **execution slots** of the **TaskTrackers**
  - The **JobTracker** considers the **localization** of the **data** when assigning the **map tasks** to the **TaskTrackers**.
  - The **JobTracker** also creates **reduce tasks** and assigns them to the TaskTrackers
  - The number of **reduce tasks** is predetermined by the **user.**
  - There is **no locality** consideration in assigning them.

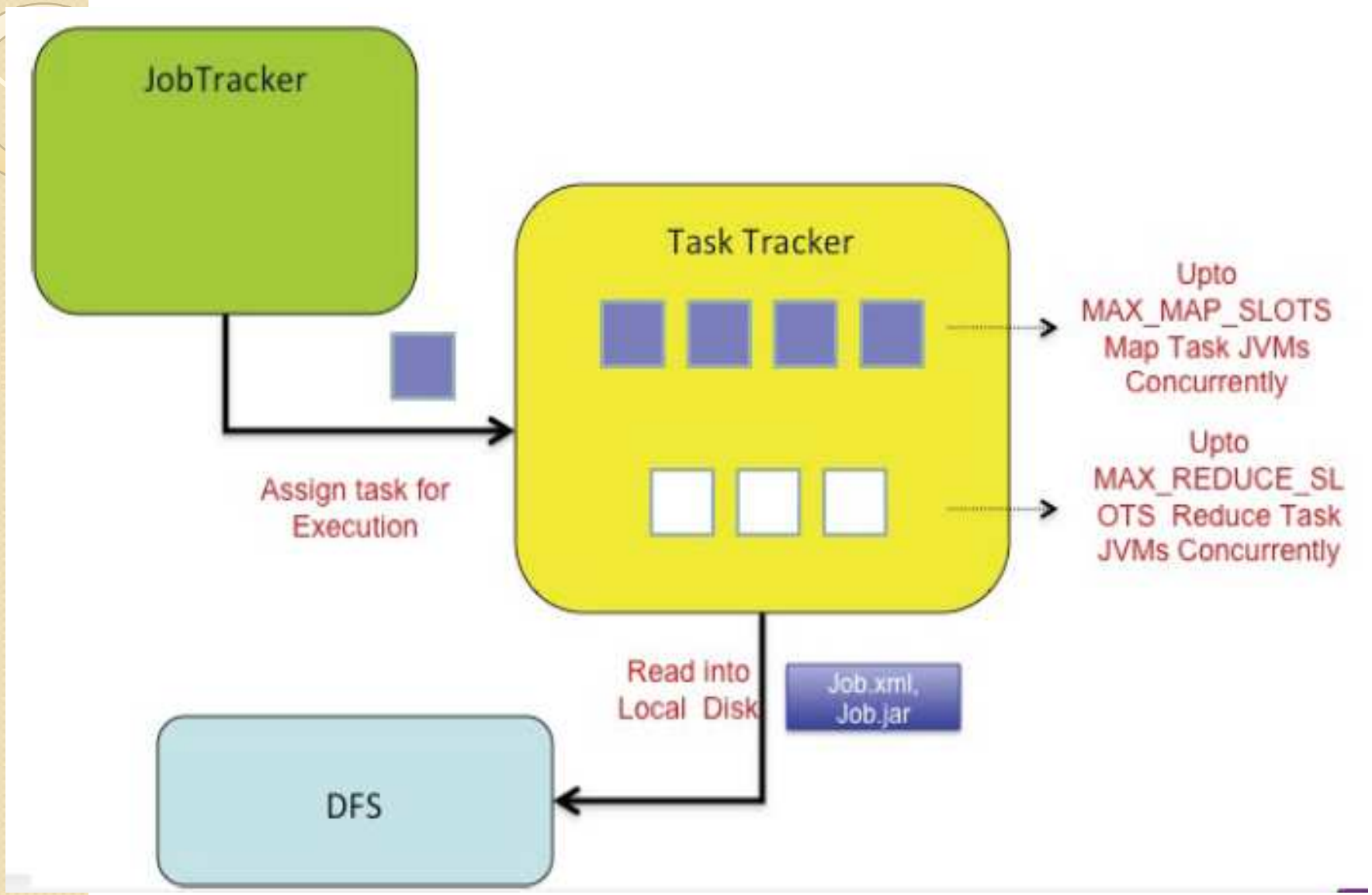
Job Scheduling



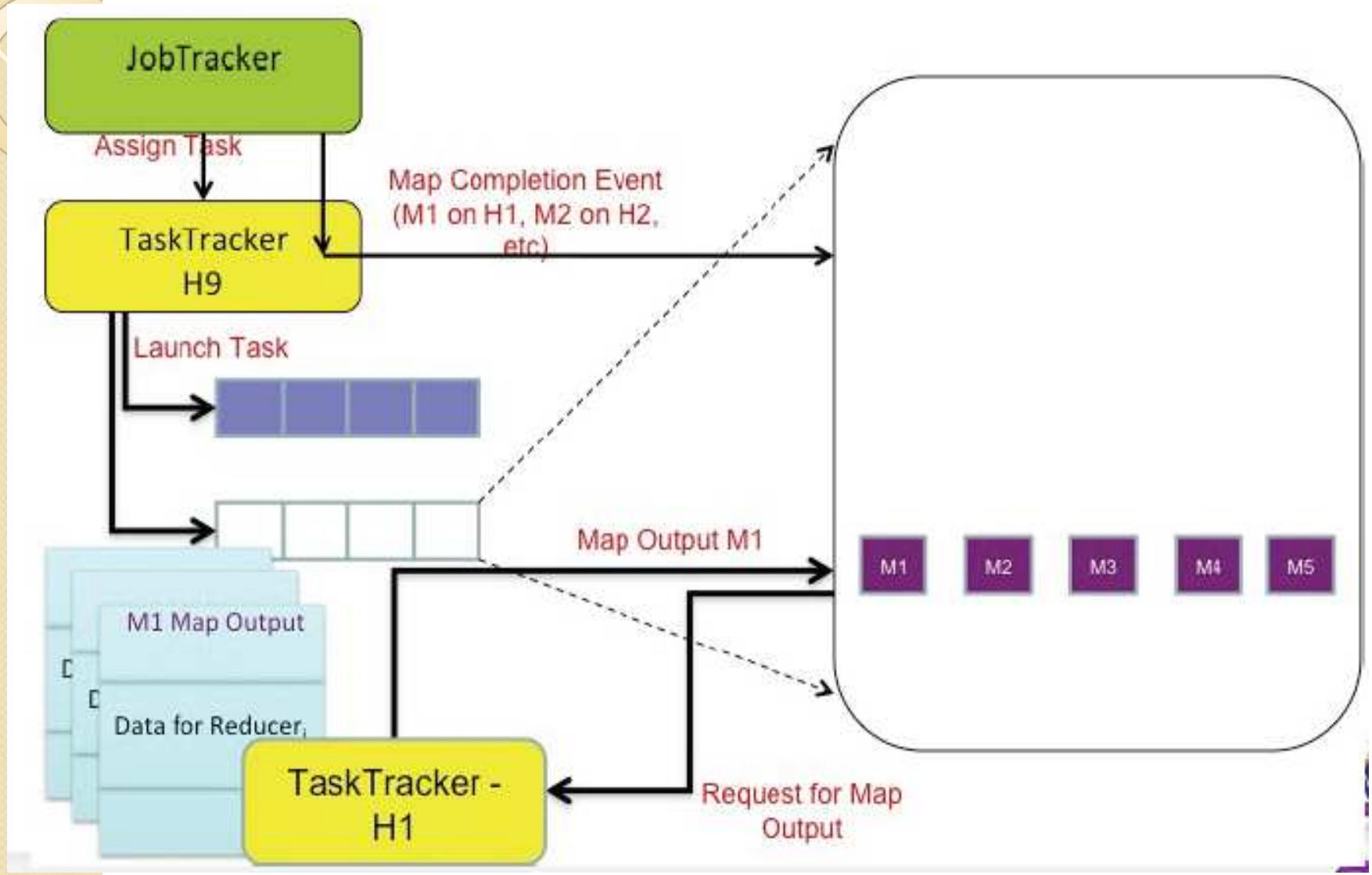
Running a Job in Hadoop

- **Task Execution**
  - The control flow to **execute** a **task** (either map or reduce) **starts** inside the **TaskTracker** by **copying** the job **JAR** file to its **file system**.
  - Instructions inside the **job JAR** file are executed after **launching** a **Java Virtual Machine (JVM)** to run its **map** or **reduce** task.

Job Execution (Map Task)



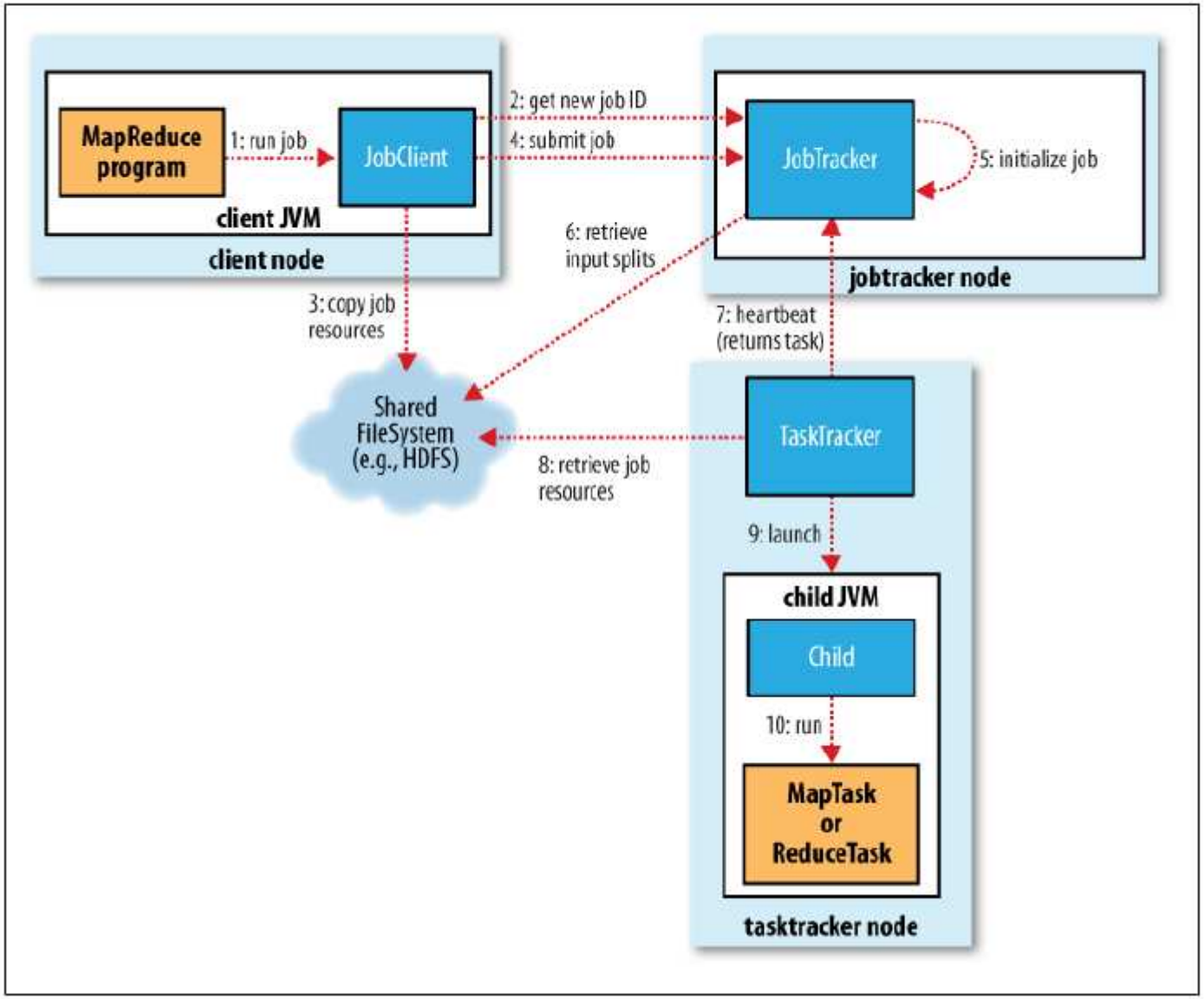
Job Execution (Reduce Task)



Running a Job in Hadoop

- **Task running check**
  - A **task running check** is performed by **receiving** periodic **heartbeat messages** to the **JobTracker** from the **TaskTrackers**.
  - Each heartbeat notifies the **JobTracker** that the sending **TaskTracker** is **alive**, and whether the sending **TaskTracker** is **ready** to **run** a **new task.**

How Hadoop runs a MapReduce job?



Challenges

Cheap nodes fail, especially if you have many

1. Mean time between failures for 1 node = 3 years
2. Mean time between failures for 1000 nodes = 1 day
3. Solution: Build fault-tolerance into system

Commodity network = low bandwidth

1. Solution: Push computation to the data

Programming distributed systems is hard

1. Solution: Data-parallel programming model: users write "map" & "reduce" functions, system distributes work and handles faults

Summary

- Process flow in MapReduce
- Motivation for Programming Paradigm
- MapReduce
  - MapReduce Operational Steps
- Twister : Iterative MapReduce
- Hadoop
  - HDFS Architecture
  - Read and Write Operations in HDFS
  - MapReduce Architecture in Hadoop
  - Running a job in Hadoop
  - Challenges

References

- Hadoop: http://hadoop.apache.org/core/
- http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
- https://www.packtpub.com/books/content/hdfs-and-mapreduce

THANK YOU