HADOOP ARCHITECTURE

CLOUD COMPUTING

Dr. Atm Shafiul Alam

a.alam@qmul.ac.uk

Queen Mary University of London

School of Electronic Engineering and Computer Science

Thanks to Dr. Gareth Tyson and Dr. Félix Cuadrado who created the original version of these slides

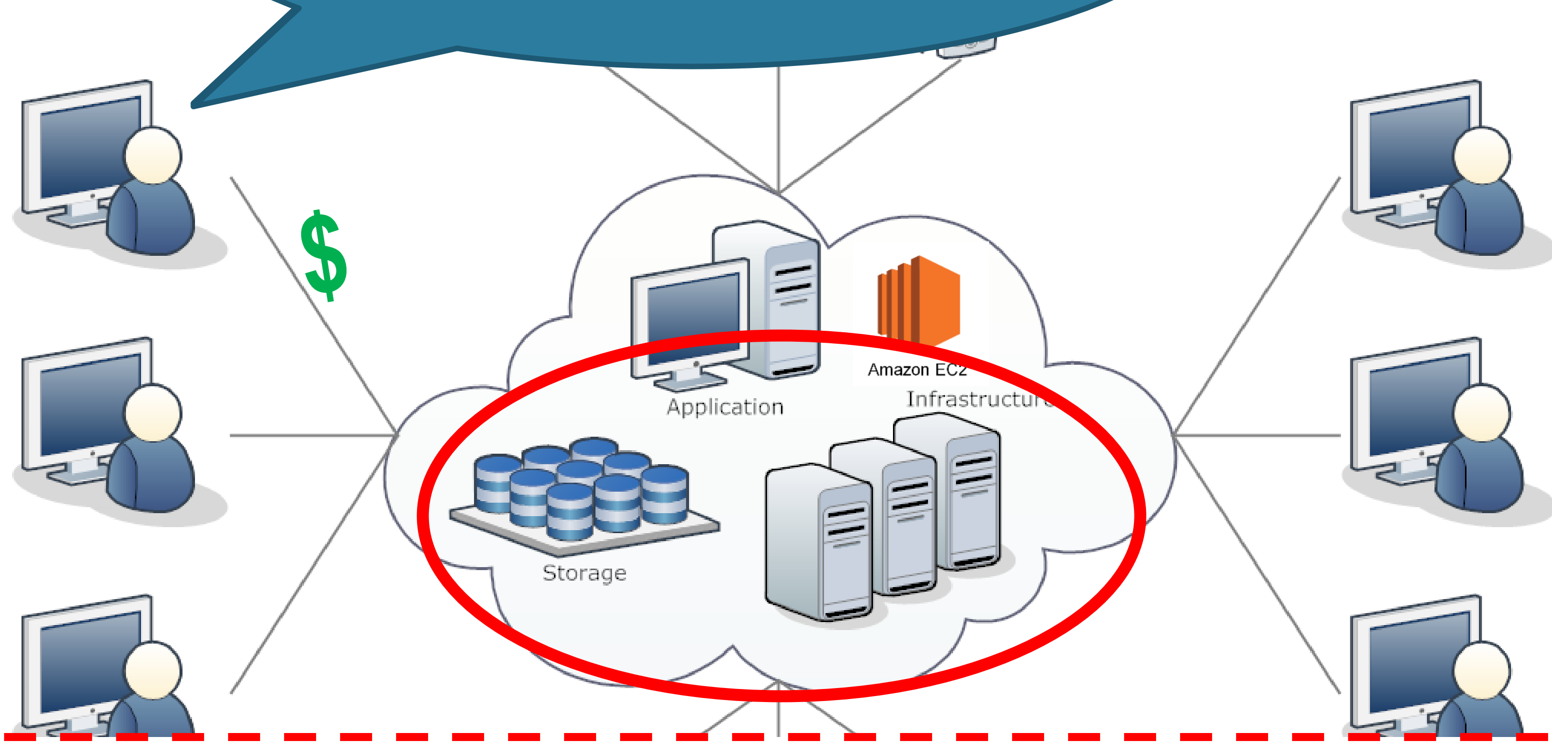
QUICK RECAP…

Yesterday we…

- Learnt about the basics of distributed computing
  - Split data up & process in parallel
- Learnt about the principles of Map Reduce
  - A particular paradigm for "Big Data"
- Ran some through basic pseudocode

What is

So I will use a "Cloud Computing" platform to hire access to machines



Cloud Computing platforms allow you to hire machines on-demand to run your tasks!

What is cloud computing?

- You have already learnt more about how cloud computing works in Week 1.
- From now, we can simply assume that we have lots of computers to run our MapReduce application on

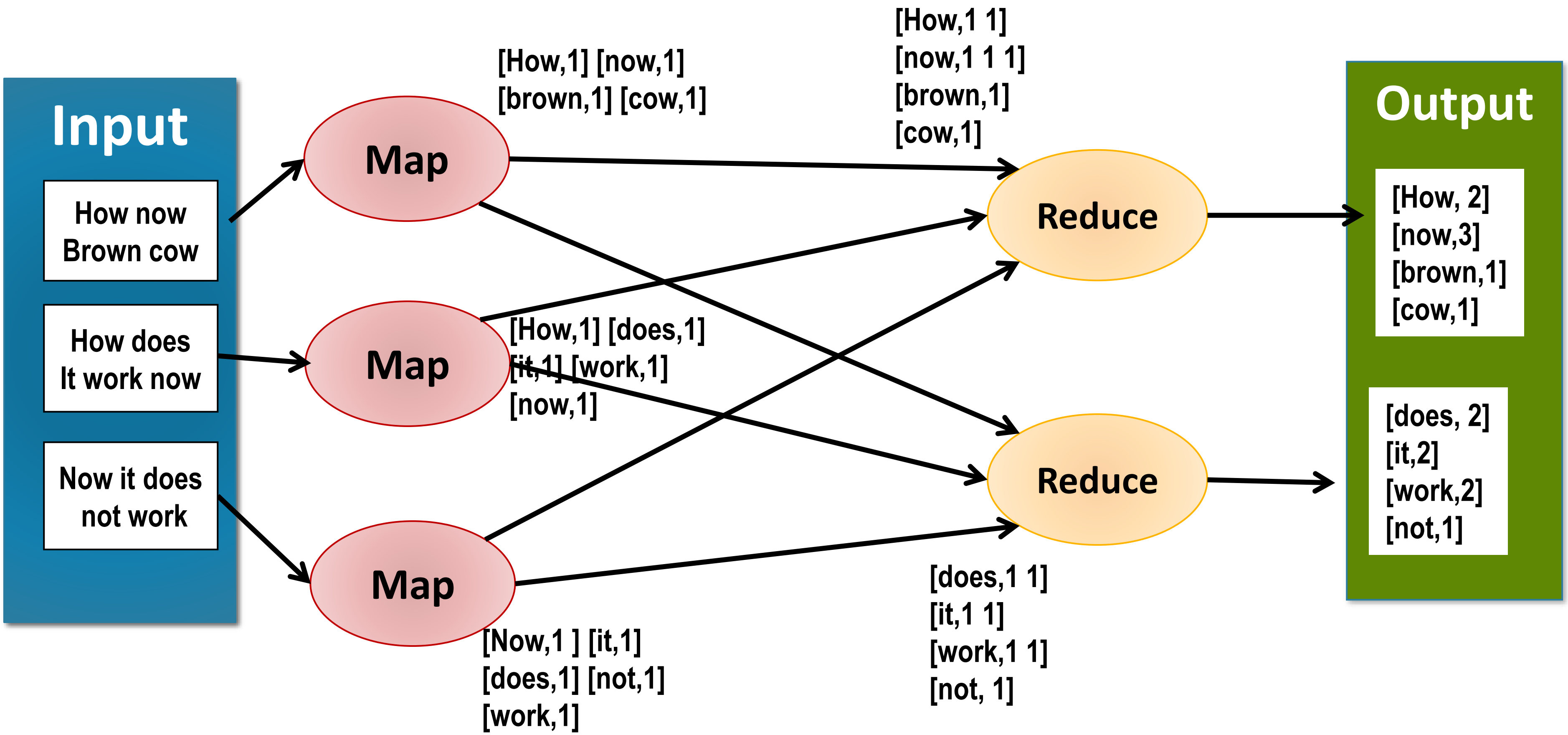
- ...but please feel free to ask if you're confused!

Parallel computing

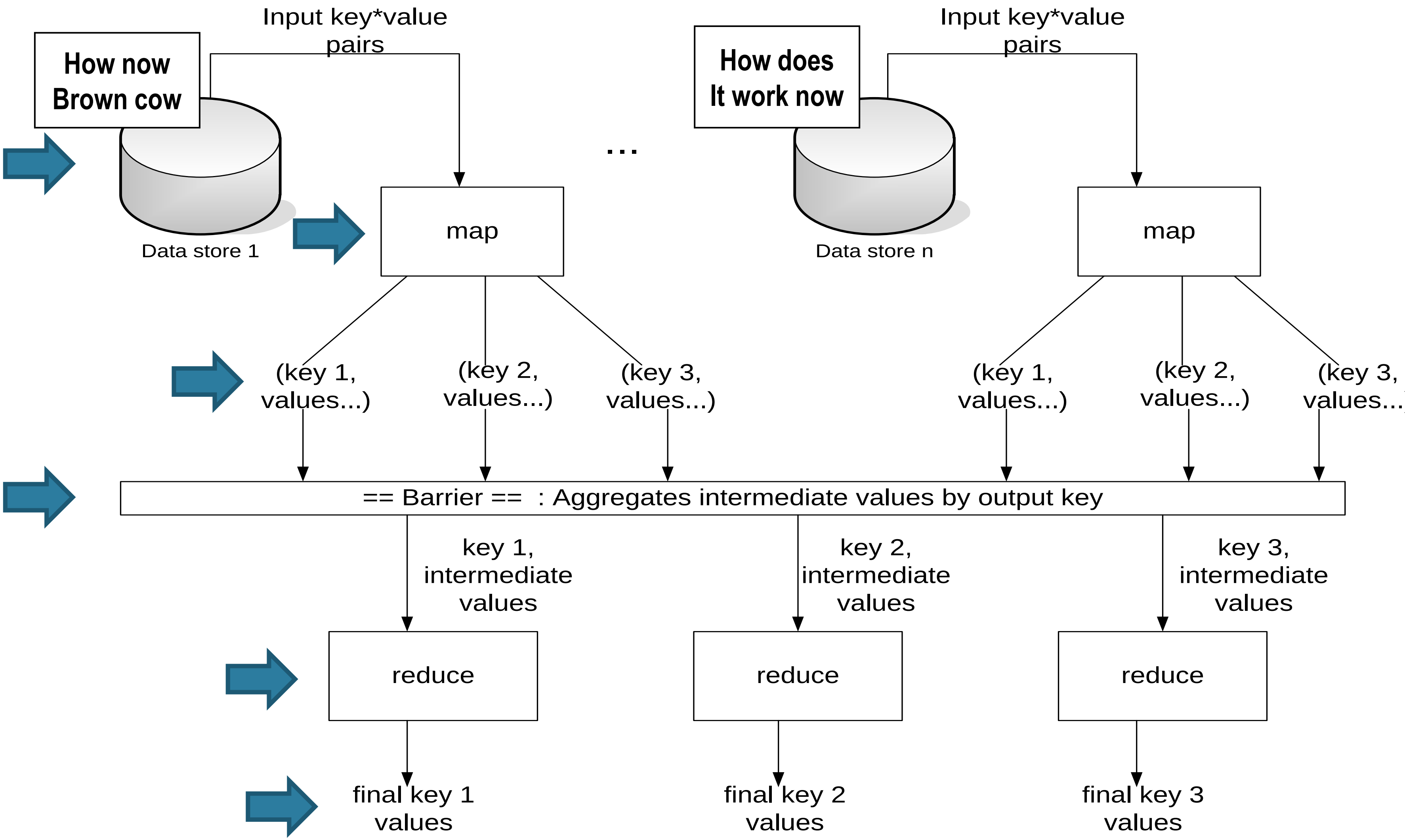
- The use of a number of **processors** to perform a calculation
- The calculation will be divided into tasks, sent to different processors (e.g. compute servers).
- Generally, we do this by **splitting data** across multiple **servers**

Note: cloud computing platforms are often used for parallel computing (because you can cheaply hire many computers)…

Word Count Example



Synchronisation and message passing



Today's contents

- **Anatomy of a MapReduce job**
- The Combiner
- Apache Hadoop
- Hadoop job execution: YARN
- Hadoop storage: HDFS

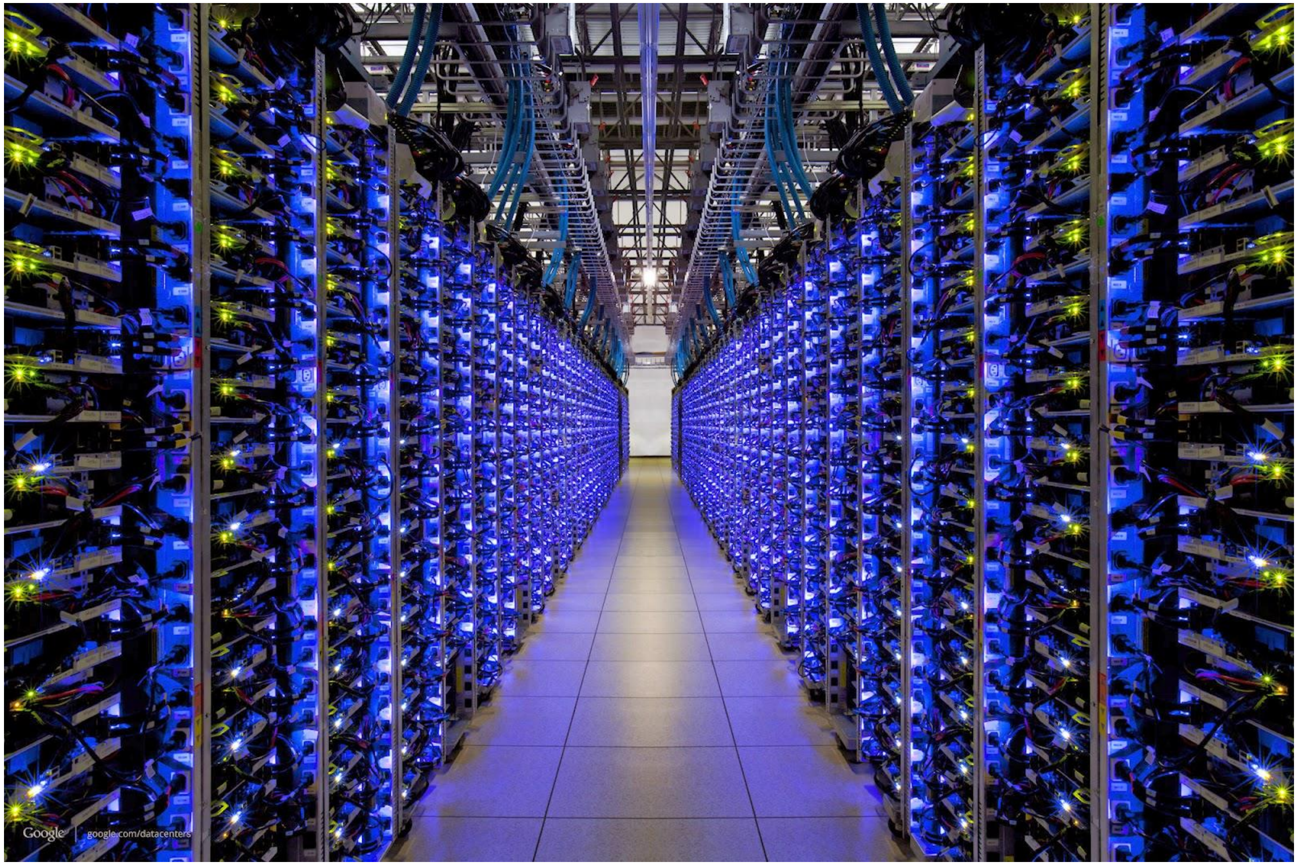
The Apache Hadoop project

- The brainchild of Doug Cutting (Yahoo)
- Apache open source framework (written in Java)
- Started in 2007 when code was spun out of Nutch
- Has grown into a large top-level project at Apache with significant ecosystem
  - V2 (YARN, 2013) structures it as a generic platform
  - Even third-party distros *a la* Linux (Cloudera, Hortonworks)

Hadoop Physical Requirements

- Designed to run in **clusters** of commodity PCs
  - Leverages heterogeneous capabilities
- Scales up to thousands of connected machines
- Suitable for Local Networks / Data Centers
  - Rack servers connected over a LAN
  - Clusters distributed over the Internet are not feasible
    - Network would become an enormous bottleneck (imagine sending terabytes over data over your DSL connection)

Hadoop Cluster



Principles of Hadoop Design

- *Data is distributed* around network
  - No centralized data server
  - Every node in cluster can host data
  - Data is replicated to support fault tolerance
- *Computation is sent to data*, rather than vice-versa
  - Code to be run is sent to nodes
  - Results of computations are aggregated as tend
- *Basic architecture is master/worker*
  - Master, *aka* JobNode, launches application
  - Workers, *aka* WorkerNodes, perform bulk of computation

Hadoop offers

- Redundant, Fault-tolerant data storage
- Parallel computation framework
- Job coordination

Hadoop offers

- Redundant, Fault-tolerant data storage
- Parallel computation framework
- Job coordination

Programmers

No longer need to worry about

Q: Where file is located?

Q: How to handle failures & data lost?

Q: How to divide computation?

Q: How to program for scaling?

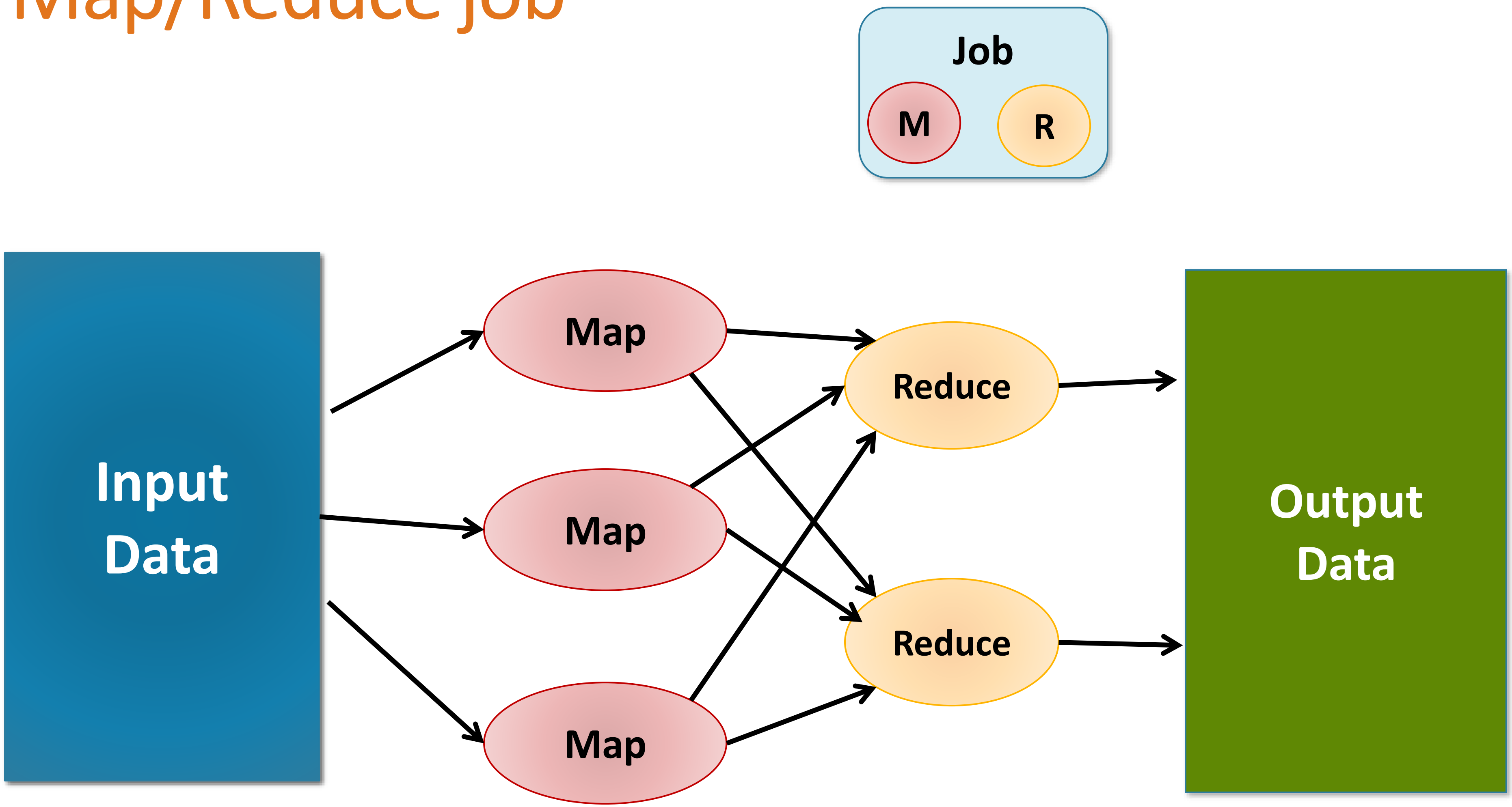
Some MapReduce Terminology

- ***Job*** – A “full program” - an execution of a Mapper and Reducer across a dataset
- ***Task*** – An execution of a Mapper or a Reducer on a slice of data
  - a.k.a. Task-In-Progress (TIP)
- ***Task Attempt*** – A particular instance of an attempt to execute a task on a machine

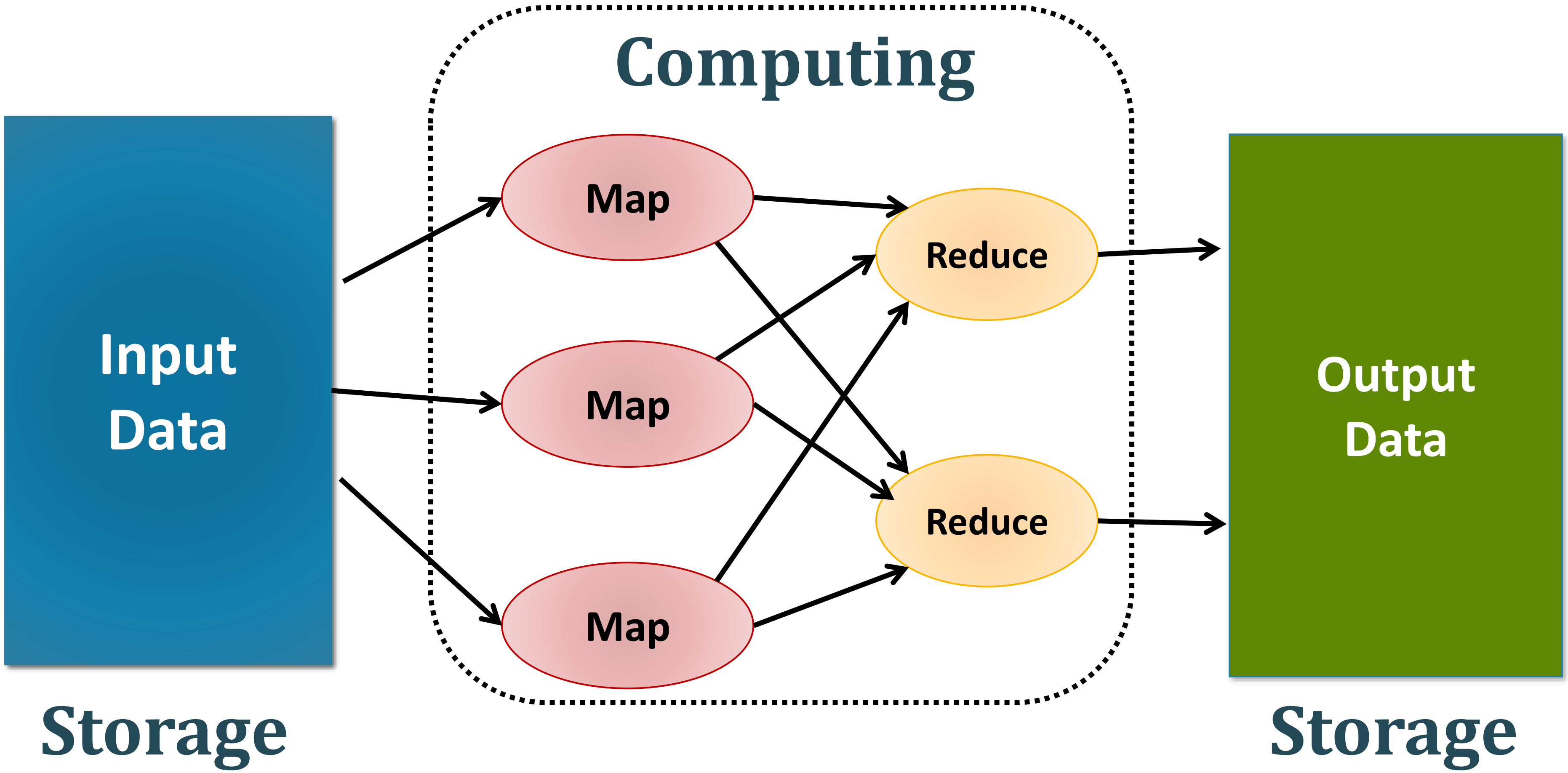
Hadoop job

- A **Hadoop Job** is packaged as a **Jar** file containing all the code for Mapper and Reducer functions
- The job is assigned a cluster-unique ID
  - A set number of reattempts is managed for job tasks
- The file is replicated over the Hadoop nodes
  - Move computation to the data

Map/Reduce job



Map/Reduce framework roles

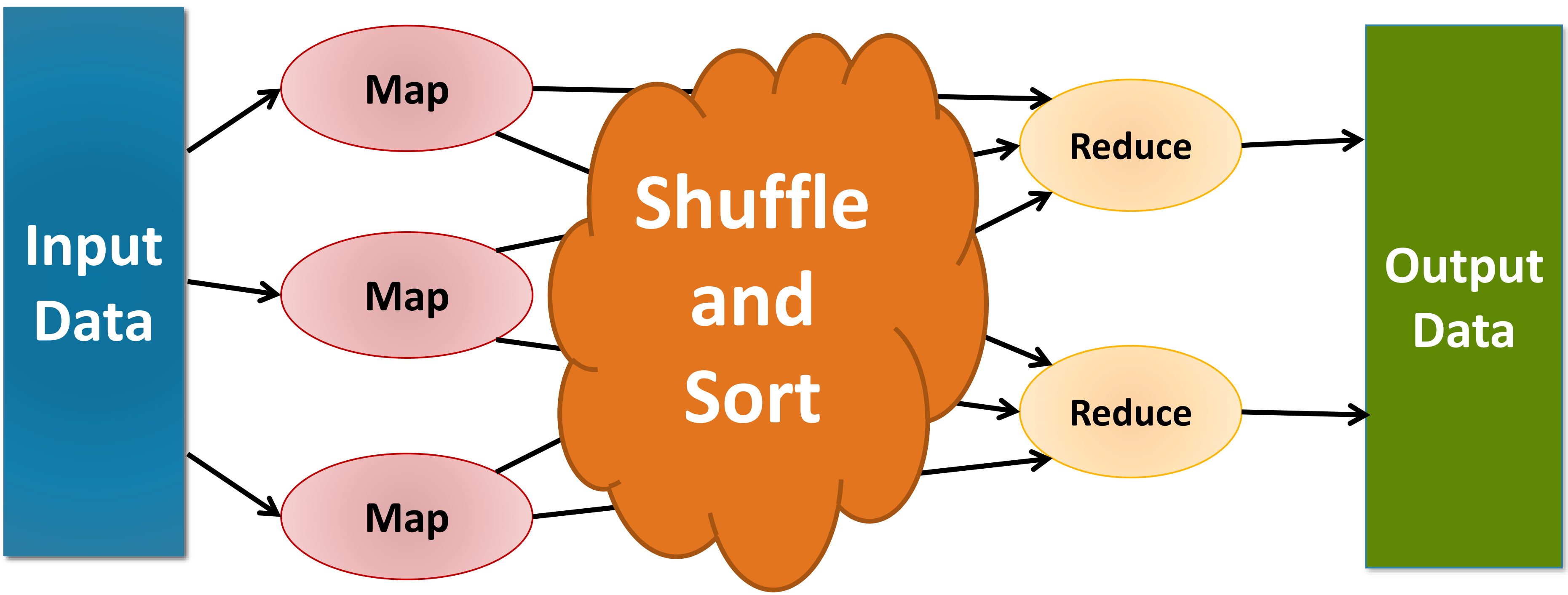


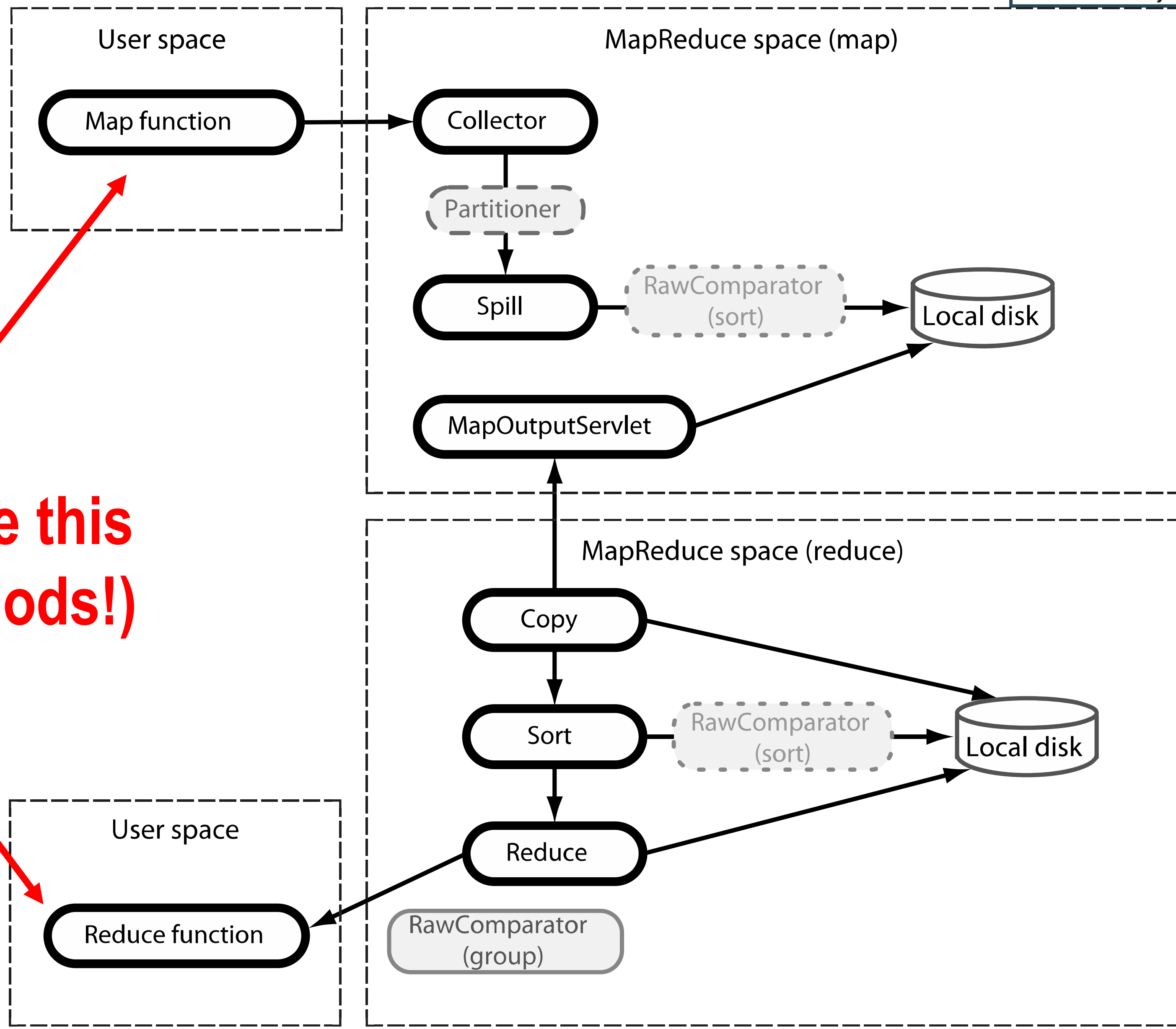
Job execution: Complete MapReduce job flow

1. Split (logically) input data into computing chunks
2. Assign one chunk to a (co-located) NodeManager
3. Run 1..* **Mappers**
4. Shuffle and Sort
5. Run 1..* **Reducers**
6. Results from Reducers create the job output

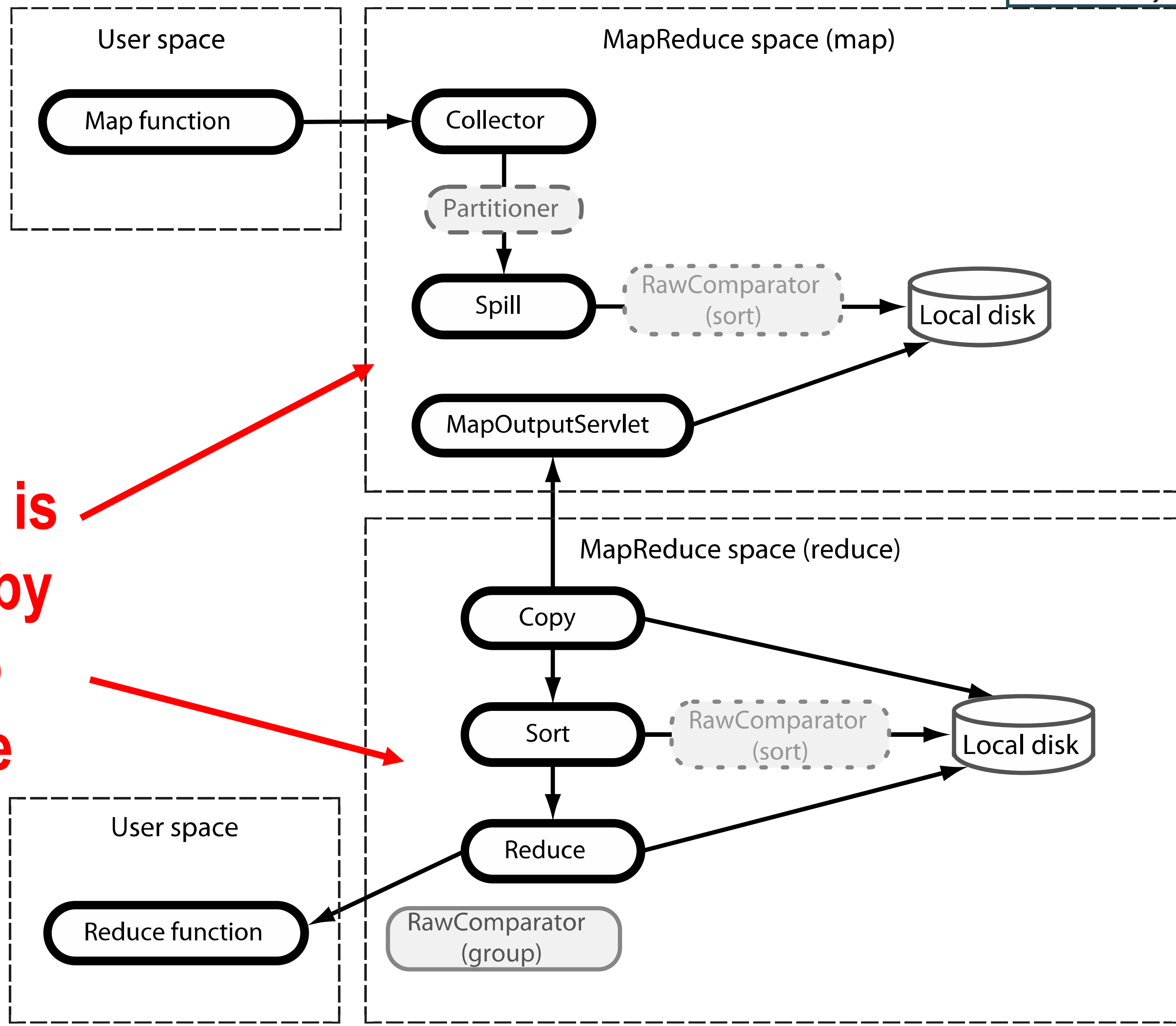
Shuffle and Sort

But we need nodes to exchange data…

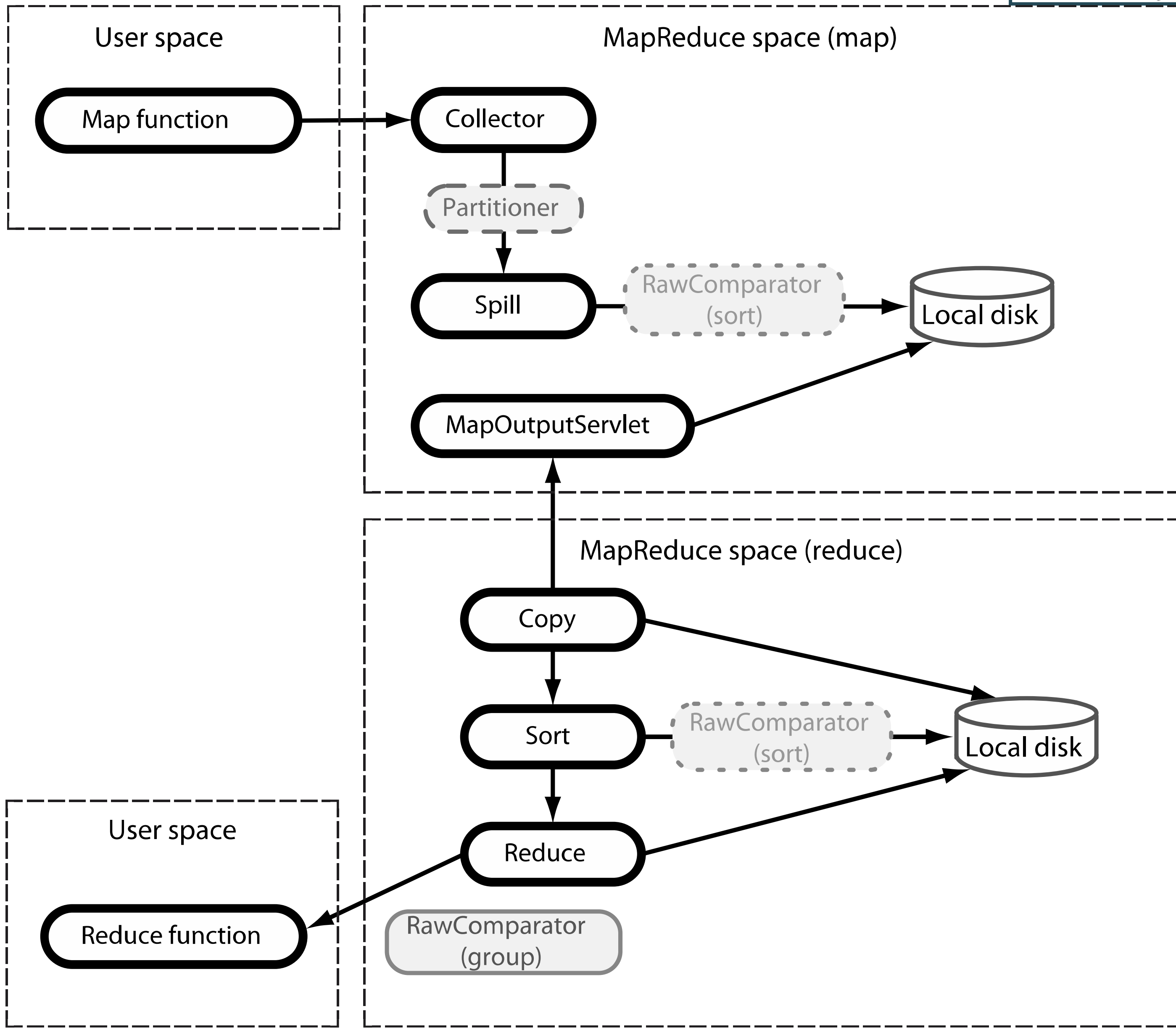




We only write this code (2 methods!)



This code is included by Hadoop software


User space
Map function
MapReduce space (map)
Collector
Partitioner
Spill
RawComparator (sort)
Local disk
MapOutputServlet
MapReduce space (reduce)
Copy
Sort
RawComparator (sort)
Local disk
Reduce
User space
Reduce function
RawComparator (group)

Let's talk about a few things...

- Shuffling
  - Moving data from mappers to reducers
- Sorting
  - Ordering outputs before being processed by reducer

Shuffling: @Mapper

1. All key value pairs are **collected**
   In-memory buffer (100MB default size), spills to HD
2. Pairs are **partitioned** depending on target reducer
   Each partition is sorted by key
3. **Combiner** runs on each partition
4. Output is available to the Reducers through HTTP server threads

Sort: @Reducer

1. The reducer **copies** output from mappers
   - Asks ApplicationManager for map output locations
2. Downloaded output is **merged** and **sorted** into the full input for the Reducer
   - List of <k2, list<v2> >, sorted by k2

Contents

The cost of communications

- Parallelizing Map and Reduce jobs allow algorithms to scale close to linearly
- One potential bottleneck for MapReduce programs is the cost of Shuffle and Sort operations
  - Data has to be copied over network communications
  - All the keys emitted by the mappers
  - Sorting large amounts of elements can be costly
- **Combiner** is an additional optional step that is executed before these steps

The Combiner

- The **combiner** acts as a **preliminary reducer**
- It is executed **at** each mapper node just before sending all the <key, value> pairs for shuffling
- Reduces the number of emitted items
  - Improves efficiency
- It *cannot* be mandatory (the algorithm must work correctly if the Combiner is not invoked)

Word count combiner

```
public void Combine(String key,
                    List<Integer> values) {
  int sum = 0;
  for (Integer count: values){
    sum+=count;
  }
  emit(key, sum);
}
```

Remind you of anything?

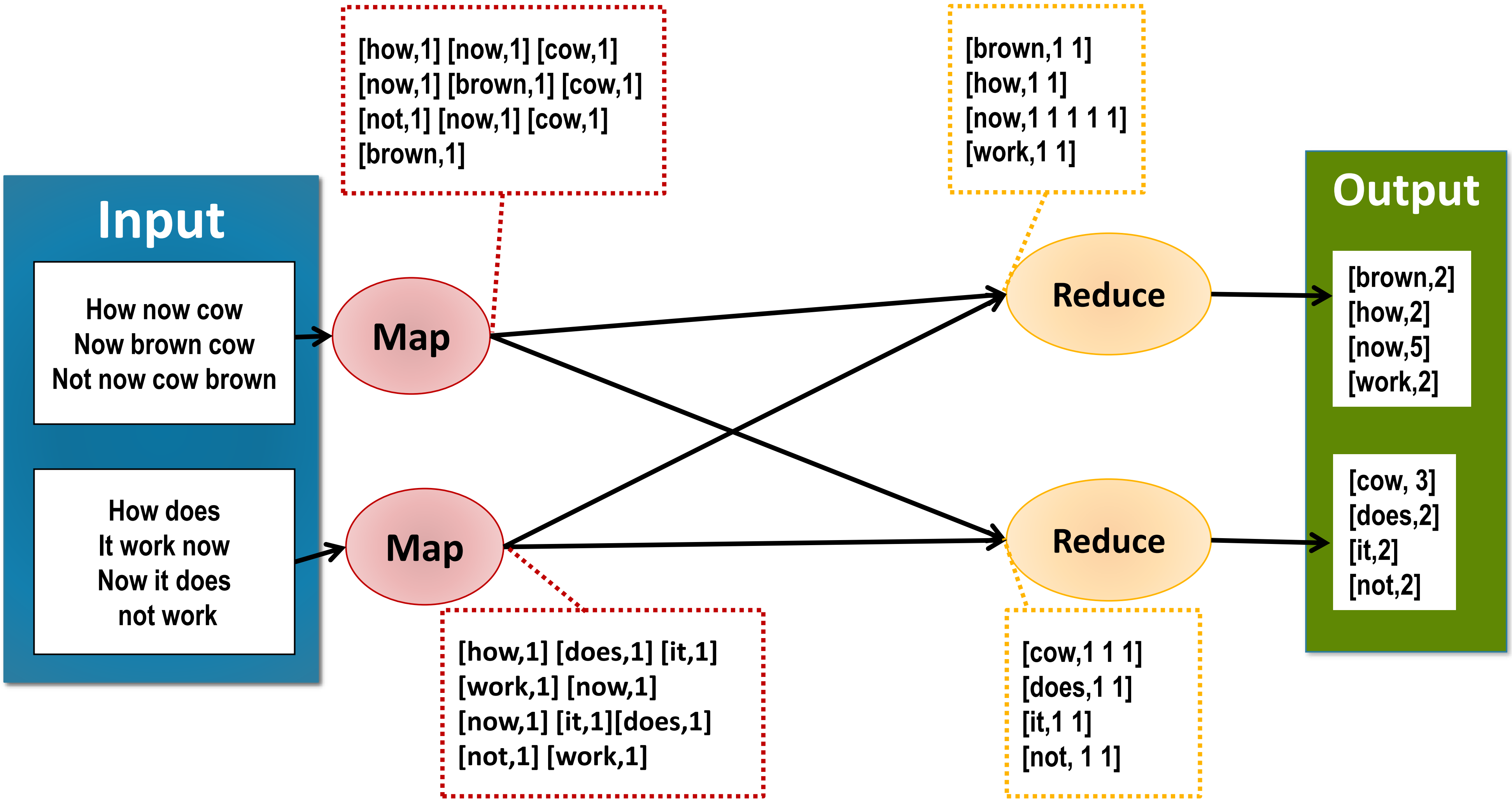
It's the same as yesterday's reducer code!

Combiner rules

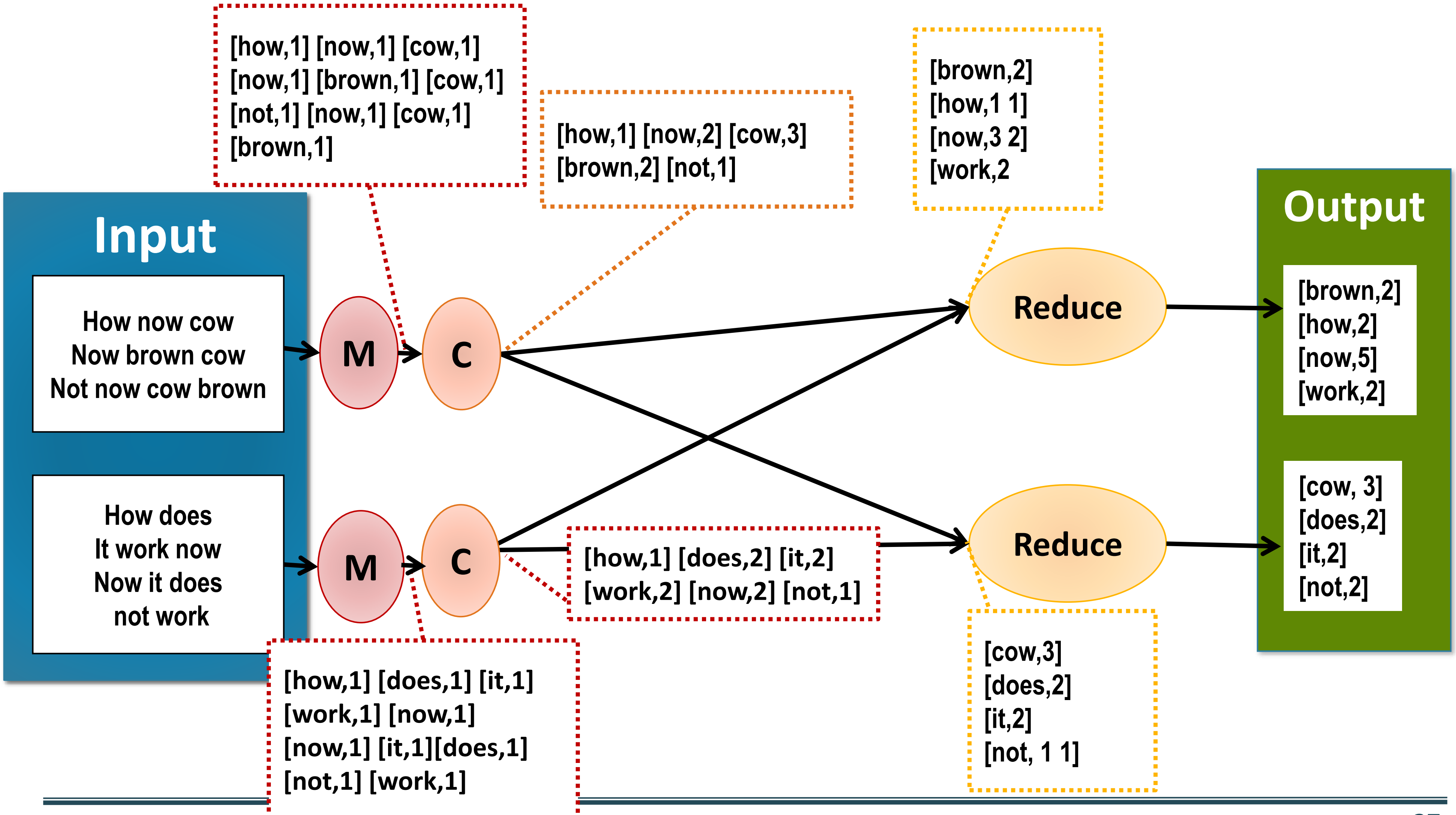
- The combiner has the same structure as the reducer (same method signature) but must comply with these rules

1. Idempotent - The number of times the combiner is applied can't change the output
2. Transititive - The order of the inputs can't change the output
3. Side-effect free - Combiners can't have side effects (or they won't be idempotent).
4. Preserve the sort order - They can't change the keys to disrupt the sort order
5. Preserve the partitioning - They can't change the keys to change the partitioning to the Reducers

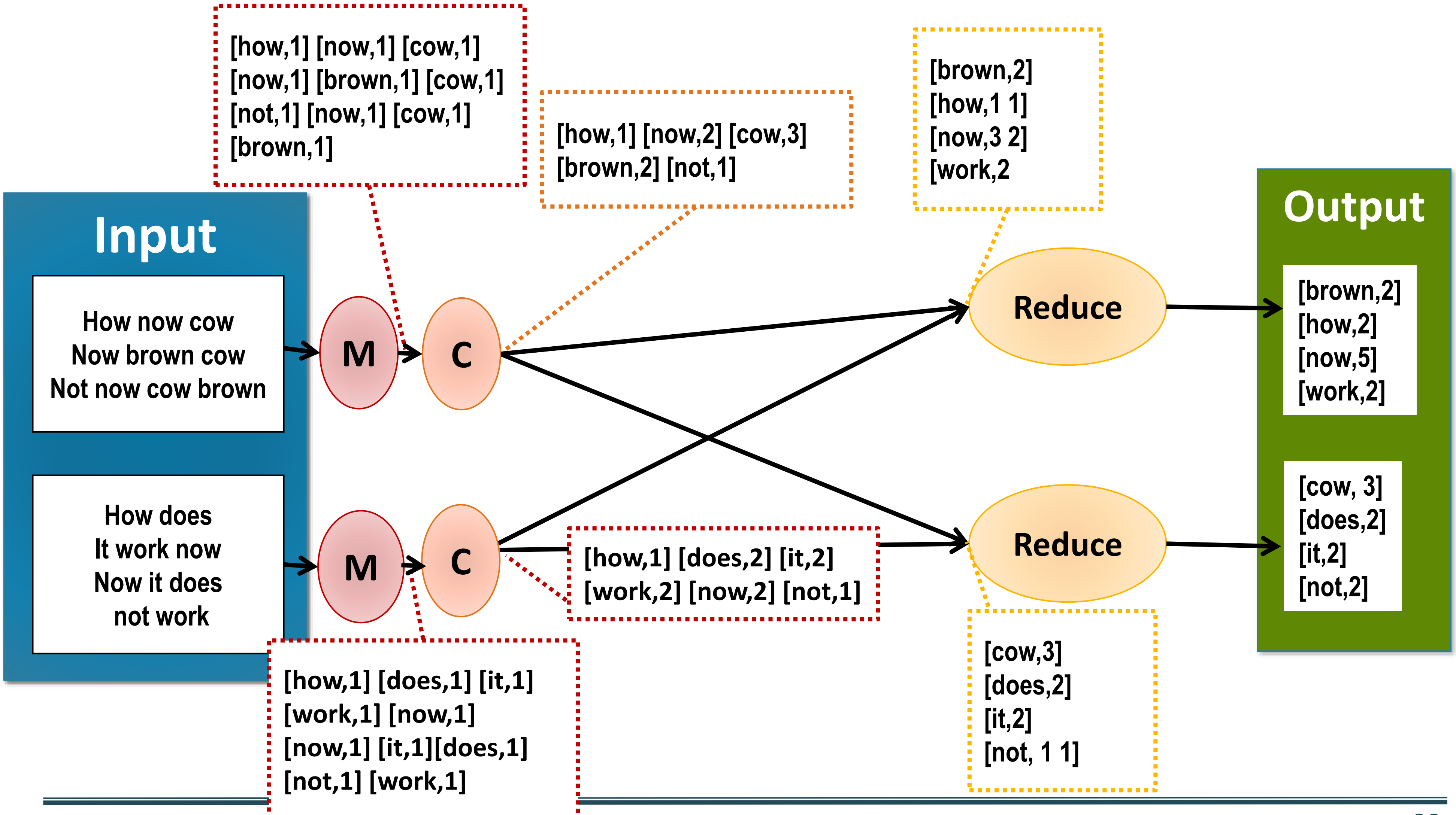
Word Count Example



Word Count Example with Combiner



Word Count Example with Combiner



Contents

- Anatomy of a MapReduce job
- The Combiner
- **Apache Hadoop**
- Hadoop job execution: YARN
- Hadoop storage: HDFS

The Apache Hadoop project

- Open source project hosted at Apache
- Implements the Map Reduce concept
- Written in Java
- Cloudera QuickStart VM includes it

Hadoop Architecture

- Hadoop executes on a cluster of networked PCs
- Each node runs a set of daemons
  - ResourceManager
  - NodeManager

Computing

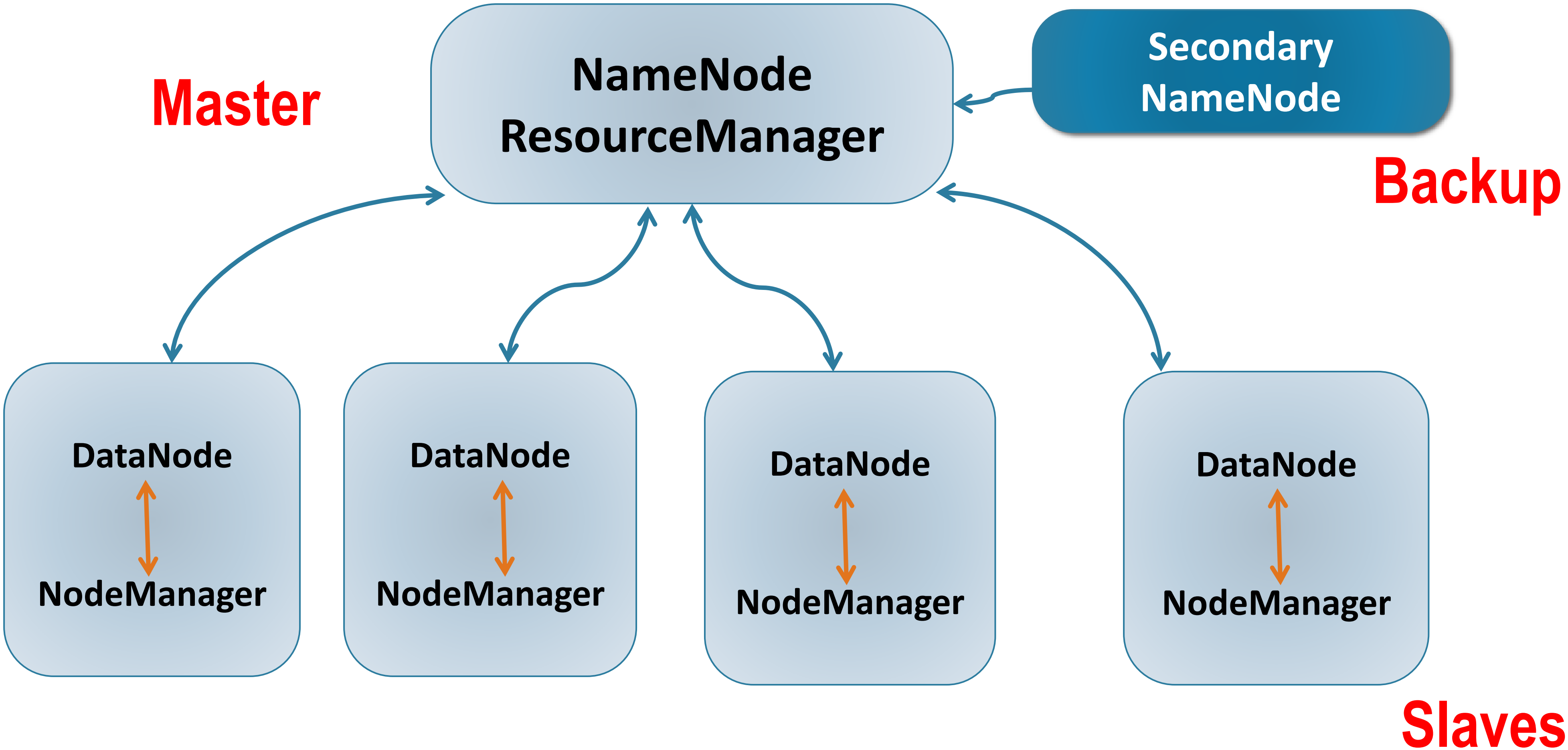
  - NameNode
  - SecondaryNameNode
  - DataNode

Storage

Master Slave Architecture

- Master (just 1)
  - Is aware of all the slave nodes
  - Receives external requests
  - Decides who executes what, and when
  - Speaks with slaves
- Slave (1..*)
  - Worker node
  - Executes the tasks the master tells it to do

Hadoop Master-Slave architecture



Contents

- Anatomy of a MapReduce job
- The Combiner
- Apache Hadoop
- **Hadoop job execution: YARN**
- Hadoop storage: HDFS

Hadoop computation tasks

- Resource management
  - Being aware of what resources are in the cluster
  - Which resources are available now
- Job allocation
  - How many resources are needed to compute the job
  - Which nodes should execute each of the tasks
- Job execution
  - Coordinate task execution from workers
  - Make sure the job completes, deal with failures

Hadoop Job allocation

- Resource management needs to estimate **how many Map and Reduce tasks** are needed for a given job
  - Based on input dataset
  - Based on job definition
- Ideally, one different node will be allocated for each different Map/Reduce tasks
  - Otherwise multiple tasks can go to same node

Job execution: Complete MapReduce job flow

1. Split input data into computing chunks
2. Assign one chunk to a (co-located) NodeManager
3. Run 1..* **Mappers**
4. Shuffle and Sort
5. Run 1..* **Reducers**
6. Results from Reducers create the job output

How many Mappers are needed?

- Mapper parallelization:
  - Each Mapper processes a different **input split**
  - Input dataset size is known
- Number of mappers = input size / split size
  - If input has multiple small files, more Mappers can be invoked (Hadoop inefficiency)
  - If input size is 100MB and split size is 10MB…how many mappers?

How many Reducers are needed?

- Reducer parallelization
  - **Keys are partitioned** across the reducers
  - Hard to automatically estimate what is the right number
  - Too many Reducers can complicate too much shuffle and sort.
- Number of reducers = **User defined parameter**
  - Remember this for your Lab session!
  - It is in MapReduce job definition class

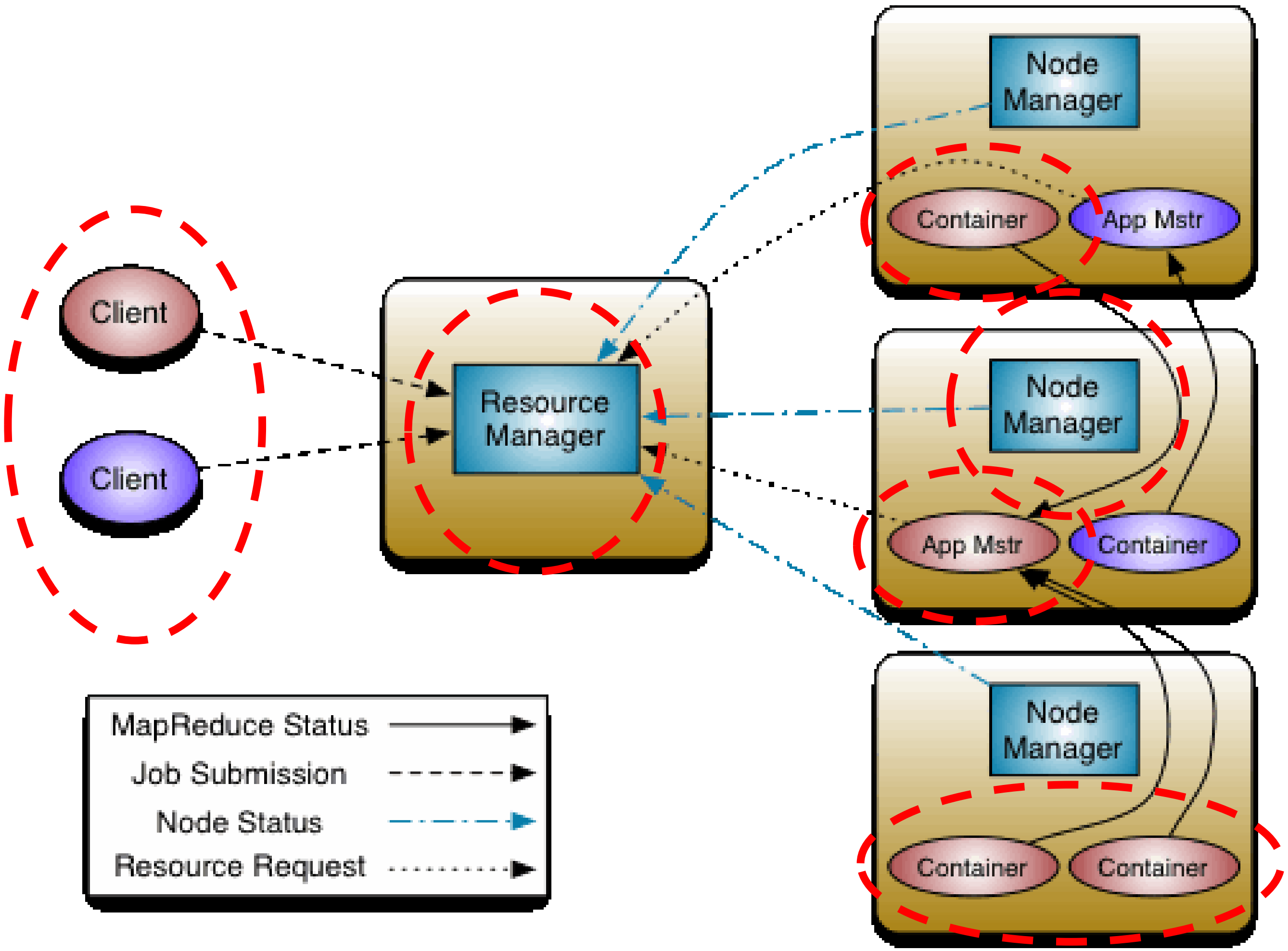
Hadoop execution daemons

- **ResourceManager** (1 per cluster)
  - Receives job requests from Hadoop Clients
  - Creates one **ApplicationMaster** per job to manage it
  - Allocates Containers in slave nodes, with assigned resources
  - Keeps track of health of NodeManager nodes
- **NodeManager** (1..* per cluster)
  - Coordinates execution of Map and Reduce tasks at node
  - Sends heartbeat messages to ResourceManager

ApplicationMaster

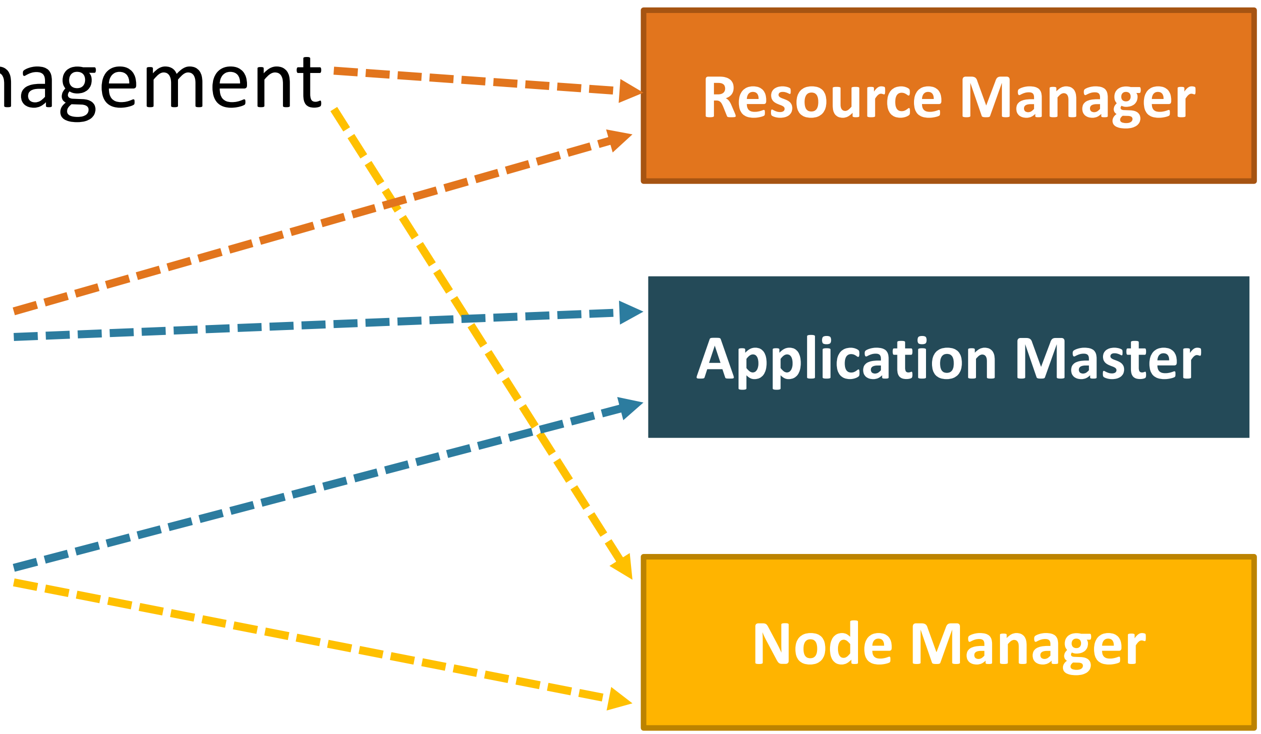
- There is one ApplicationMaster per job.
- Implements the specific computing framework (e.g., MapReduce)
  - After creation, negotiates with ResourceManager how many resources will be required for the job
  - Decides which nodes will run Map and Reduce jobs among the Containers given by the ResourceManager
  - Reports to the ResourceManager about the progress and completion of the whole job
  - Is destroyed when the job is completed

Job Execution Architecture (YARN)



Responsibilities on computation tasks

- Resource management
- Job allocation
- Job execution



Contents

- Anatomy of a MapReduce job
- The Combiner
- Apache Hadoop
- Hadoop job execution: YARN
- **Hadoop storage: HDFS**

HDFS

- HaDoop Distributed Filesystem
  - Shared storage among the nodes of the Hadoop cluster
- Storage for input and output of MapReduce jobs
- HDFS is Tailored for MapReduce jobs
  - Large block size (64MB default)
    - But not too large, blocks define the minimum parallelization unit
  - HDFS is not a POSIX compliant filesystem
    - Tradeoffs for improving data processing throughput

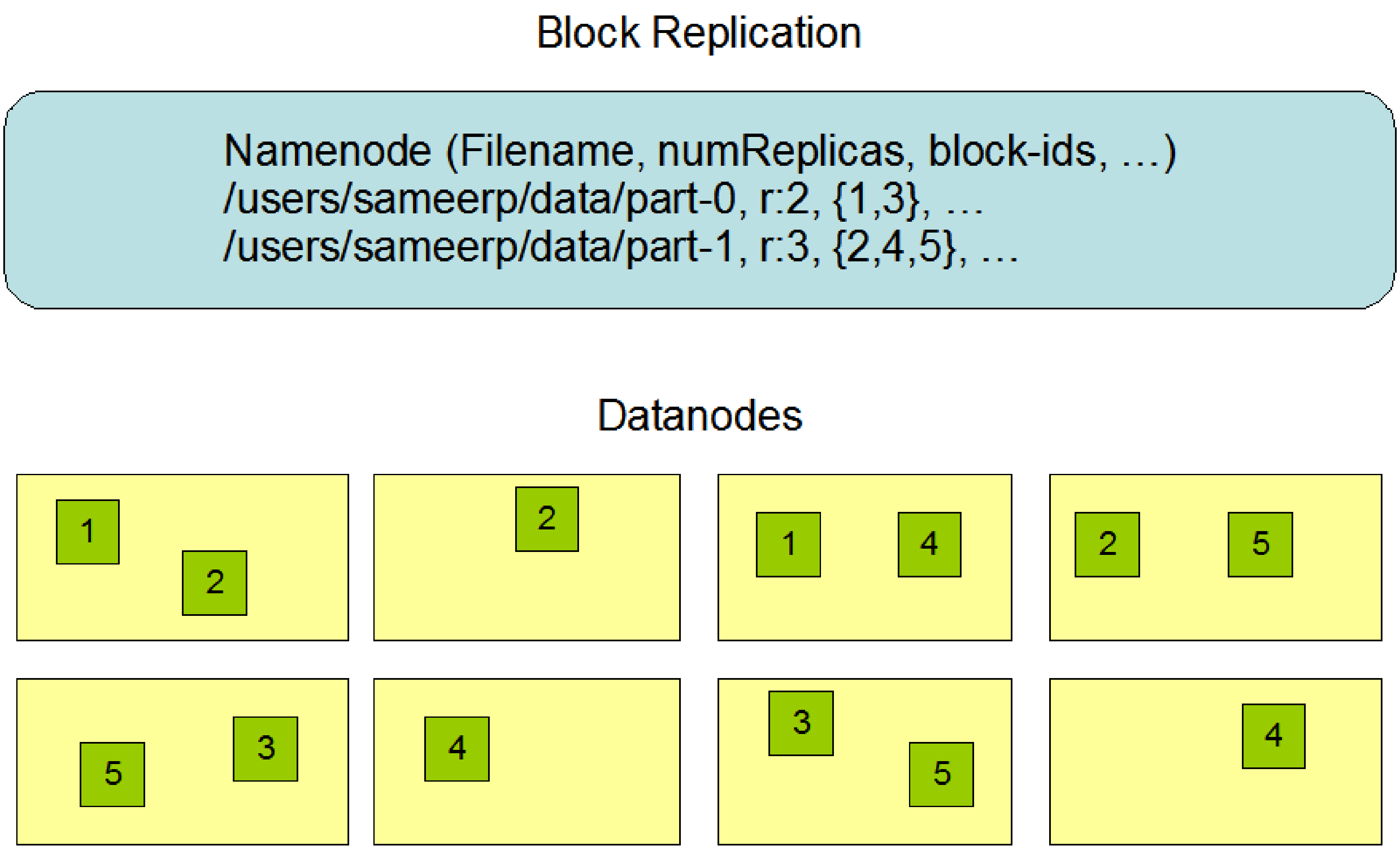
HDFS Data distribution

- Data distribution is a key element of the MapReduce model and architecture
- "Move computation to data" principle
  - Rather than copying data to the nodes that will process it, the jar is copied to the nodes where the data is stored
- Blocks are replicated over the cluster
  - Default ratio is three times
  - Spread replicas among different physical locations
  - Improves reliability

Hadoop Storage Daemons

- DataNode (1..* per cluster)
  - Stores blocks from the HDFS
  - Report periodically to NameNode list of stored blocks
- NameNode (1 per cluster)
  - Keeps index table with (all) the locations of each block
  - Heavy task, no computation responsibilities
  - Single point of failure
- Secondary Namenode (1 per cluster)
  - Communicates periodically with NameNode
  - Stores backup copy of index table

Data replication

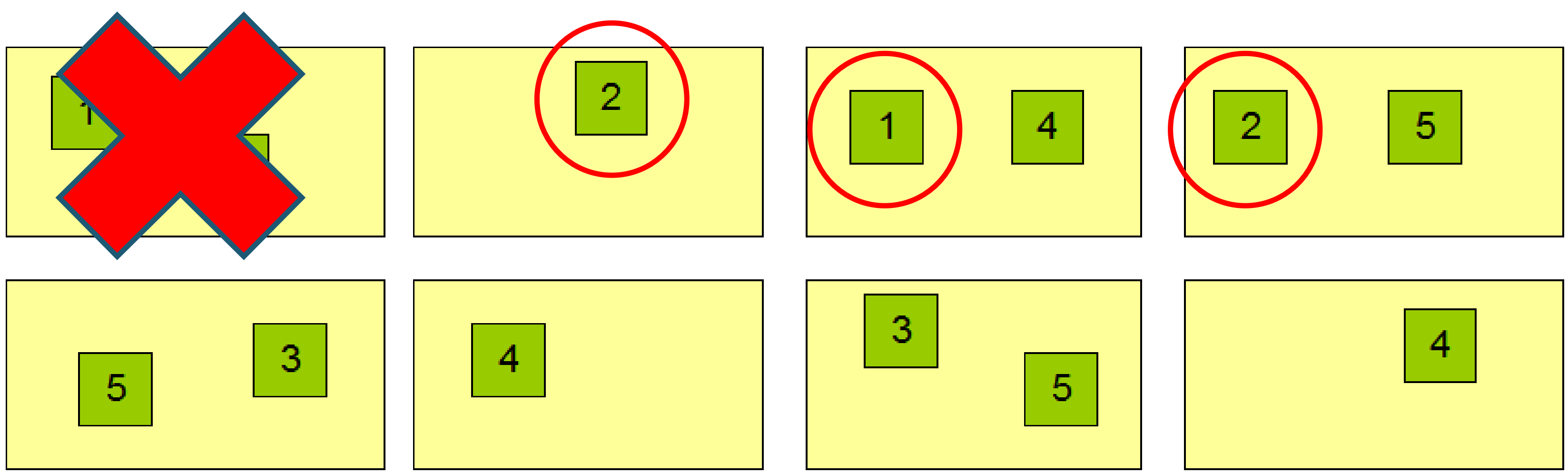


Data replication

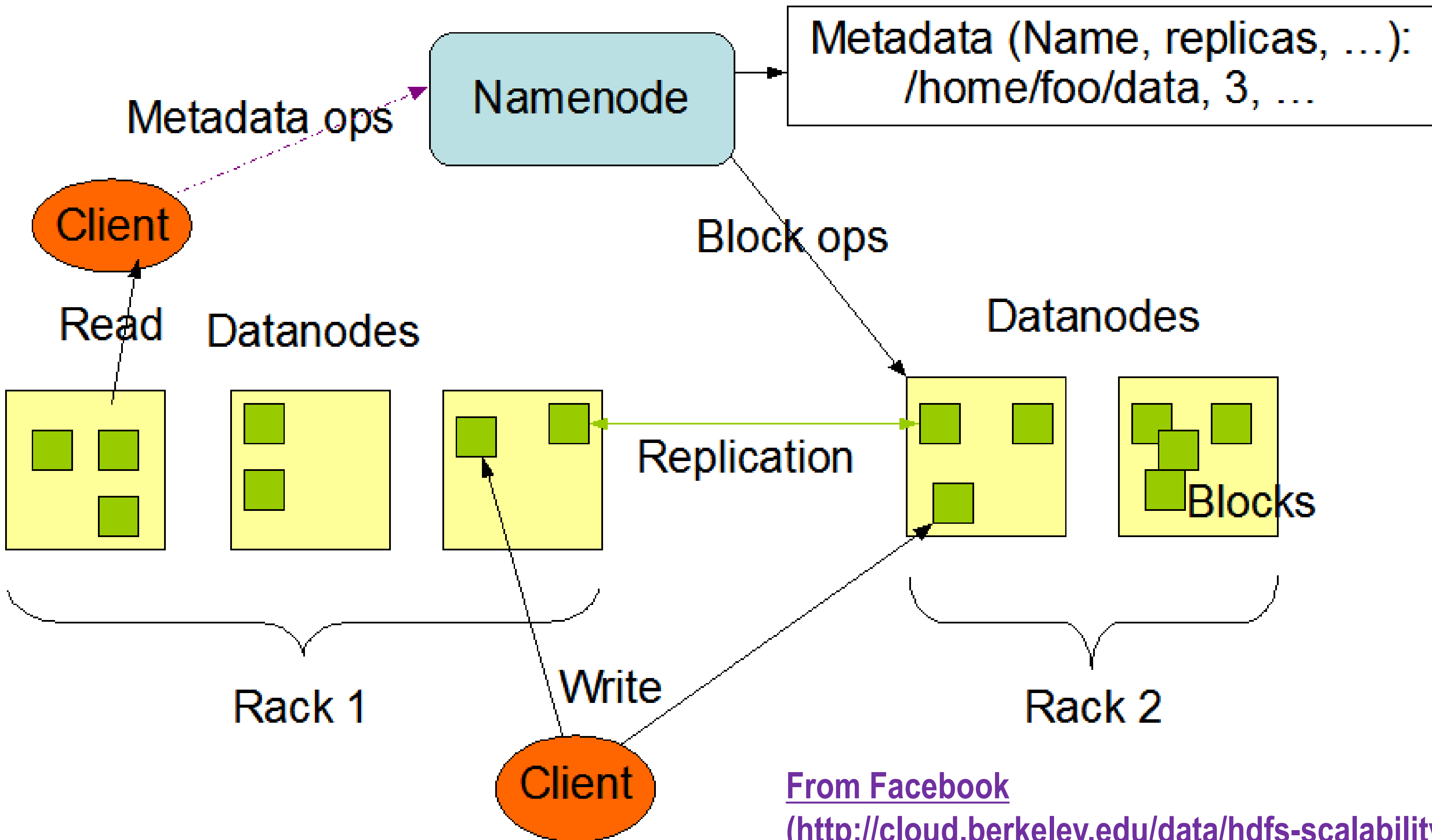
Block Replication

Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …

Datanodes

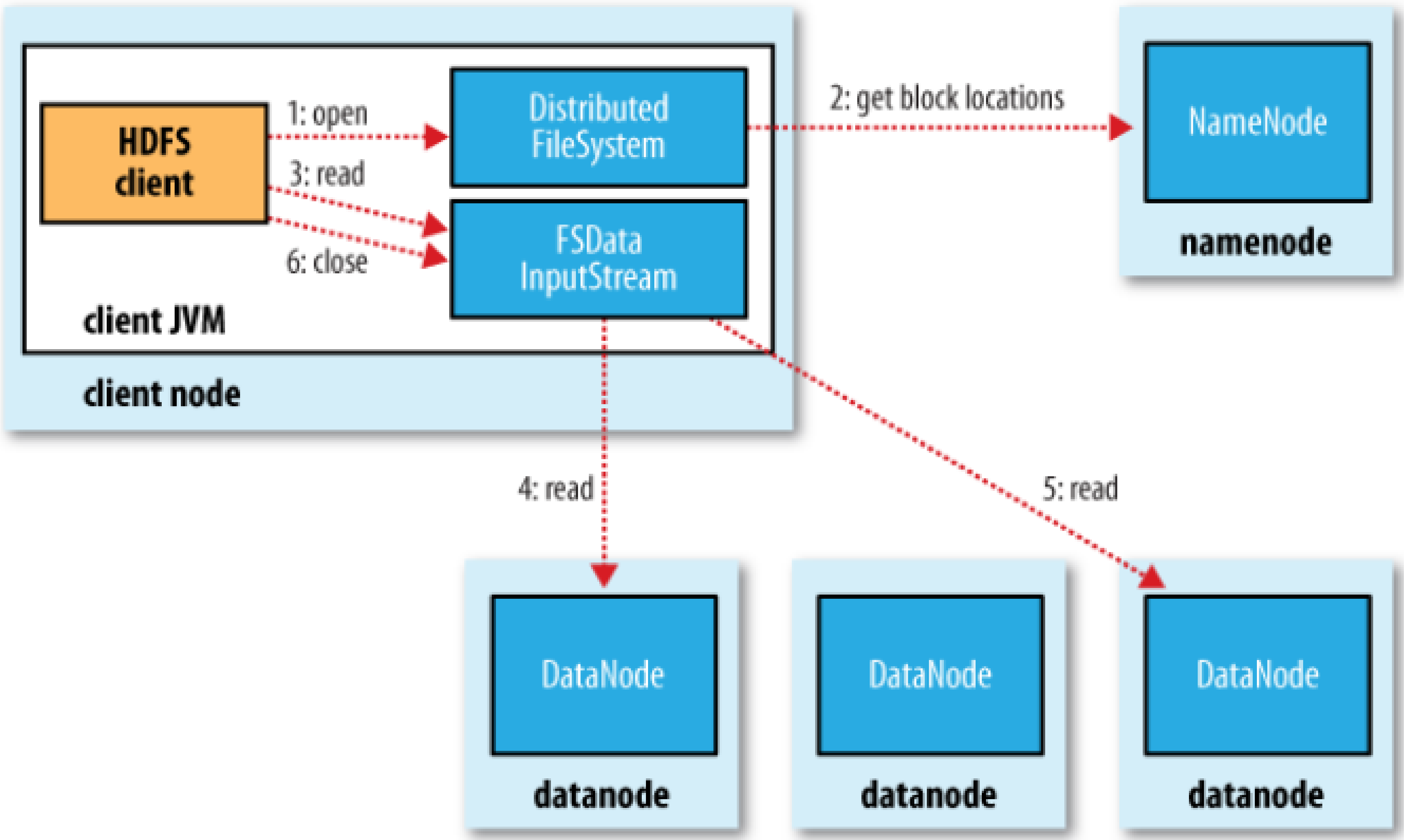


HDFS Usage

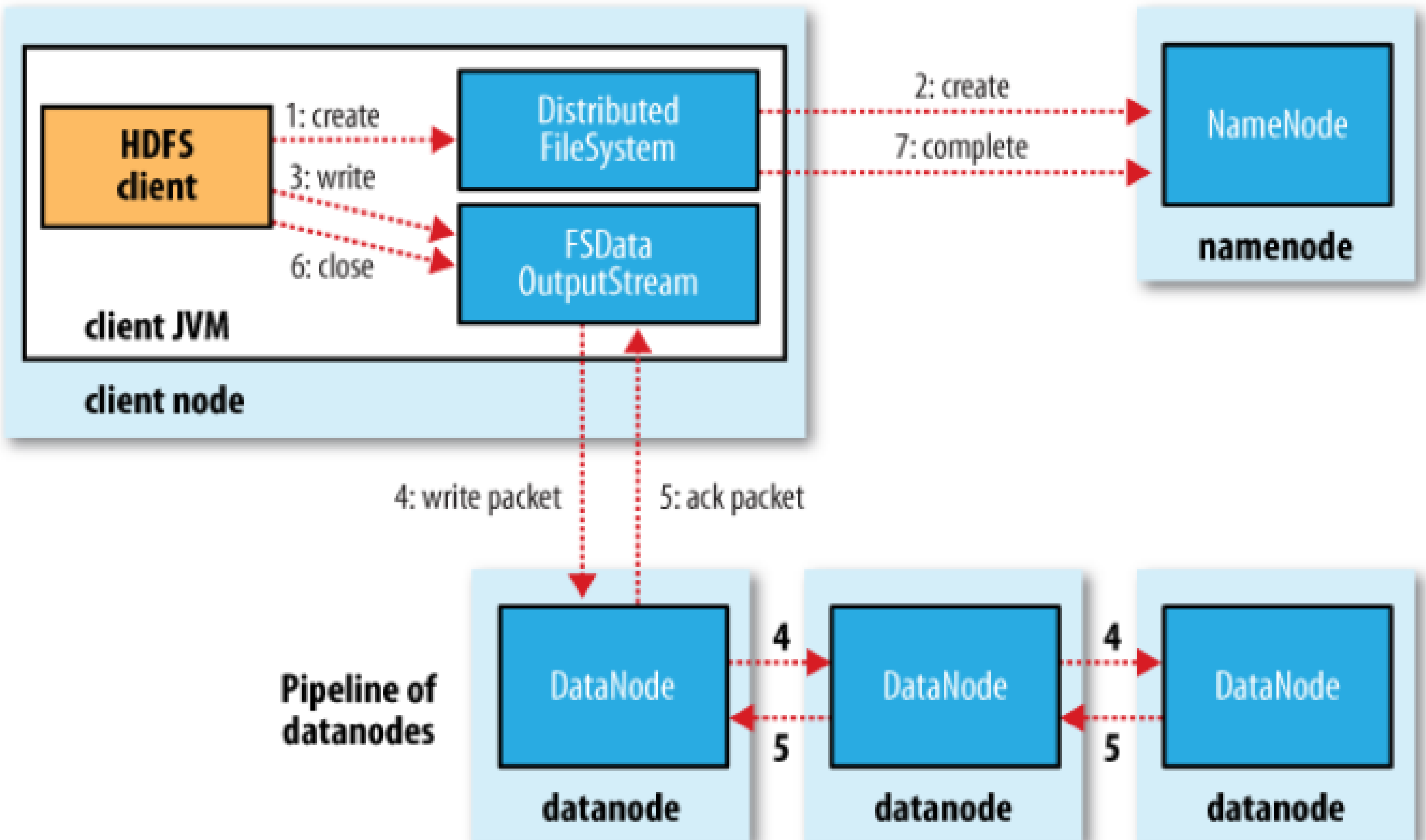


From Facebook (http://cloud.berkeley.edu/data/hdfs-scalability.pdf)

HDFS File Read operation



HDFS File Write Operation



MR Job input and output data

1. Input data -> Mappers

- Mappers are assigned input splits from HDFS input path
  - (default 64MB)
- Data locality optimization: ApplicationManager attempts to assign Mappers where data block is stored

2. Reducers -> Output data

- Reducer output copied to HDFS
  - One file per Reducer
- For reliability concerns, HDFS replication

Queen Mary University of London

Recommended reading

- Hadoop YARN: Yet Another Resource Negotiator
  - http://www.socc2013.org/home/program/a5-vavilapalli.pdf
- How MapReduce works, Chapter 6, Hadoop: The Definitive Guide, 3rd Edition.
  - Available in QMUL Safaribooksonline
- HDFS design: http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Summary

- Anatomy of a MapReduce job
- The Combiner
- Apache Hadoop
- Hadoop job execution: YARN
- Hadoop storage: HDFS