PMCA506L: Cloud Computing

Module 4 : Cloud Programming Paradigms

Courtesy : *Ming Lian , Dogules E Comer & Other Sources of Internet*

The Map Reduce Programming Paradigm

- *MapReduce* is a programming model for data processing

- The power of MapReduce lies in its ability to scale to 100s or 1000s of computers, each with several processor cores

- How large an amount of work?

  - Web-Scale data on the order of 100s of GBs to TBs or PBs

  - It is likely that the input data set will not fit on a single computer's hard drive

  - Hence, a distributed file system (e.g., Google File System- GFS) is typically required

Motivations

- Motivations
  - Large-scale data processing on clusters
  - Massively parallel (hundreds or thousands of CPUs)
  - Reliable execution with easy data access
- Functions
  - Automatic parallelization & distribution
  - Fault-tolerance
  - Status and monitoring tools
  - A clean abstraction for programmers
    - Functional programming meets distributed computing
    - A batch data processing system

Commodity Clusters

- MapReduce is designed to efficiently process large volumes of data by connecting many commodity computers together to work in parallel

- A *theoretical* 1000-CPU machine would cost a very large amount of money, far more than 1000 single-CPU or 250 quad-core machines

- MapReduce ties smaller and more reasonably priced machines together into a single cost-effective *commodity cluster*

Isolated Tasks

- MapReduce divides the workload into multiple *independent tasks* and schedule them across cluster nodes

- A work performed by each task is done *in isolation* from one another

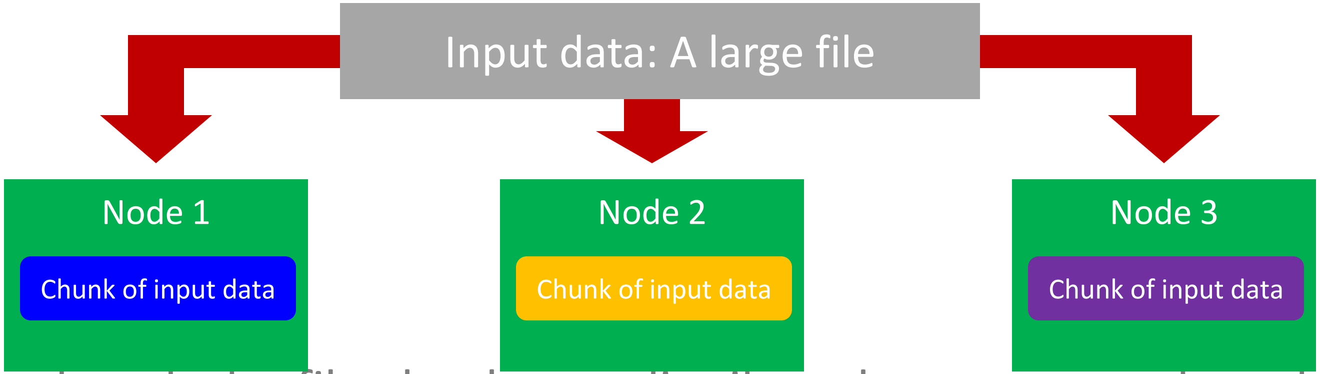
- The amount of communication which can be performed by tasks is mainly limited for scalability reasons

  - The communication overhead required to keep the data on the nodes synchronized at all times would prevent the model from performing reliably and efficiently at large scale

Data Distribution

- In a MapReduce cluster, data is distributed to all the nodes of the cluster as it is being loaded in

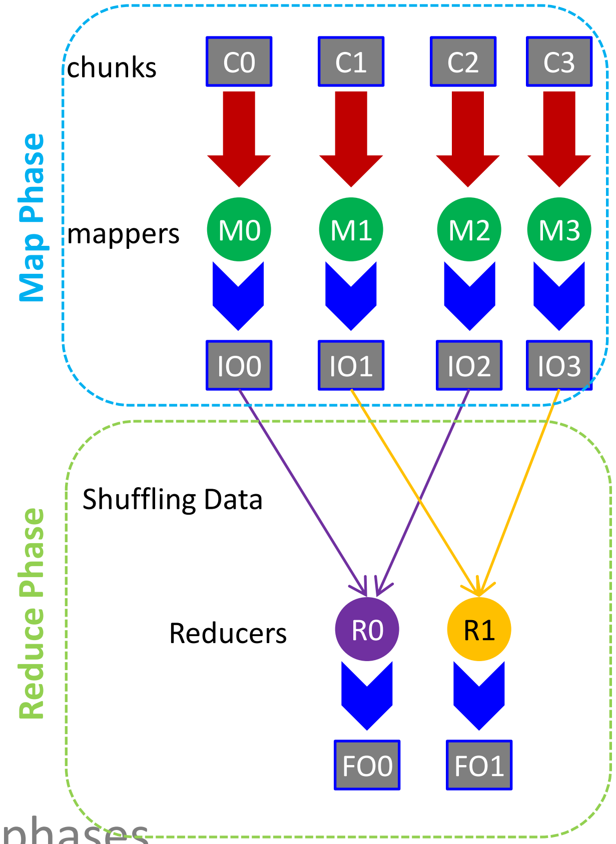
- An underlying distributed file systems (e.g., GFS) splits large data files into chunks which are managed by different nodes in the cluster



- Even though the file chunks are distributed across several machines, they form *a single namesapce*

MapReduce: A Bird's-Eye View

- In MapReduce, chunks are processed in isolation by tasks called *Mappers*
- The outputs from the mappers are denoted as intermediate outputs (IOs) and are brought into a second set of tasks called *Reducers*
- The process of bringing together IOs into a set of Reducers is known as *shuffling process*
- The Reducers produce the final outputs (FOs)
- Overall, MapReduce breaks the data flow into two phases, *map phase* and *reduce phase*

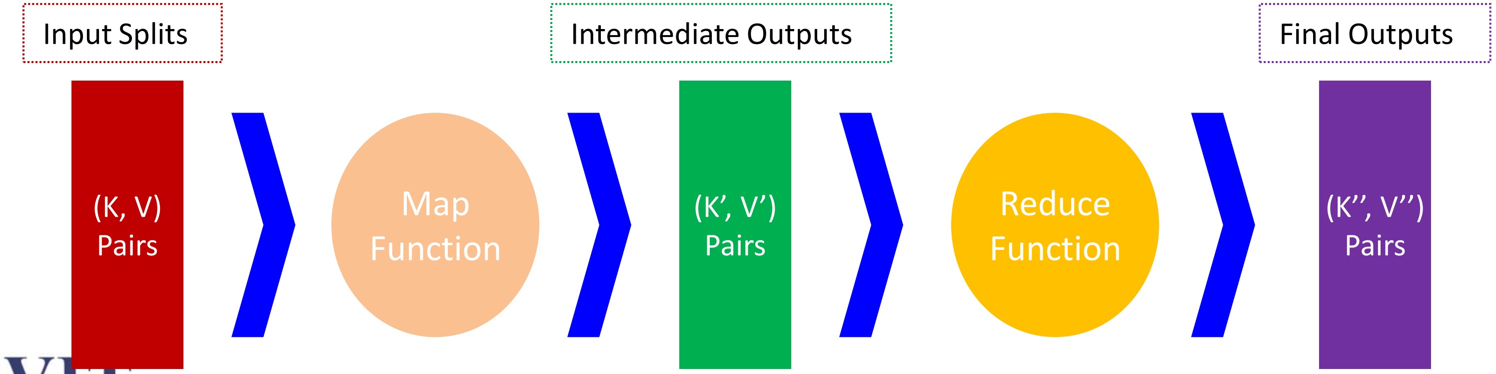


Keys and Values

- The programmer in MapReduce has to specify two functions, the *map function* and the *reduce function* that implement the Mapper and the Reducer in a MapReduce program

- In MapReduce data elements are always structured as key-value (i.e., (K, V)) pairs

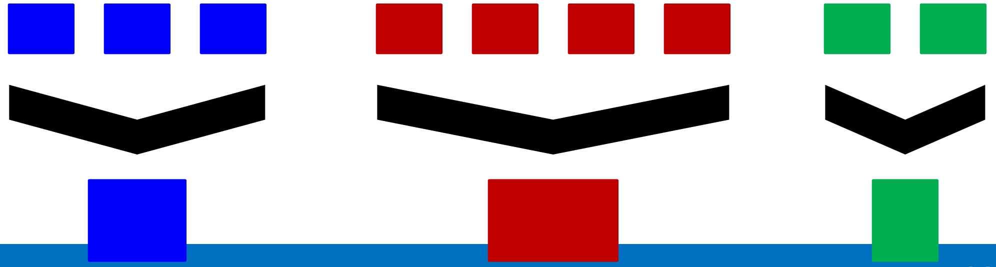
- The map and reduce functions receive and *emit* (K, V) pairs



Partitions

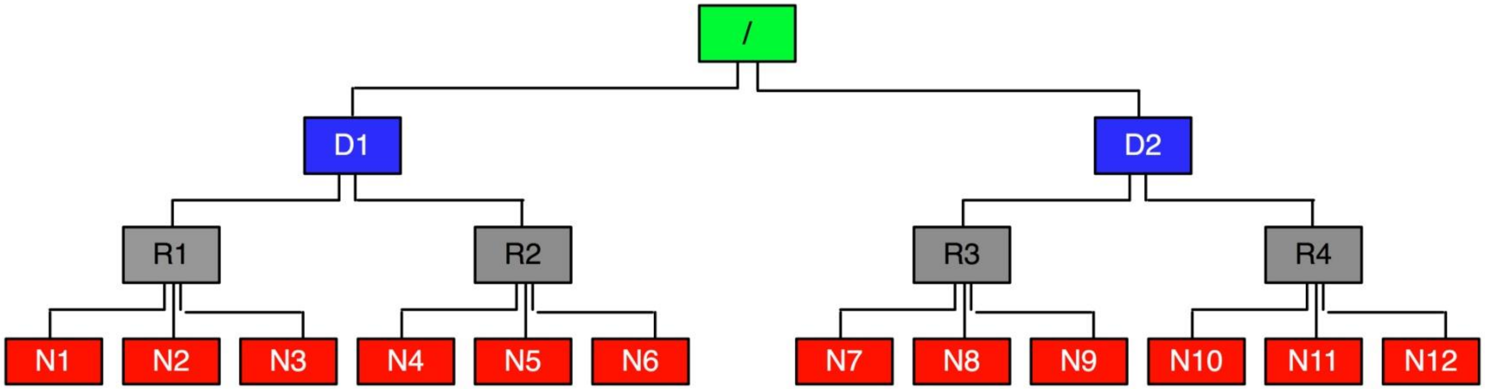
- In MapReduce, intermediate output values are not usually reduced together
- *All values with the same key are presented to a single Reducer together*
- More specifically, a different subset of intermediate key space is assigned to each Reducer
- These subsets are known as *partitions*

Different colors represent different keys (potentially) from different Mappers



Partitions are the input to Reducers

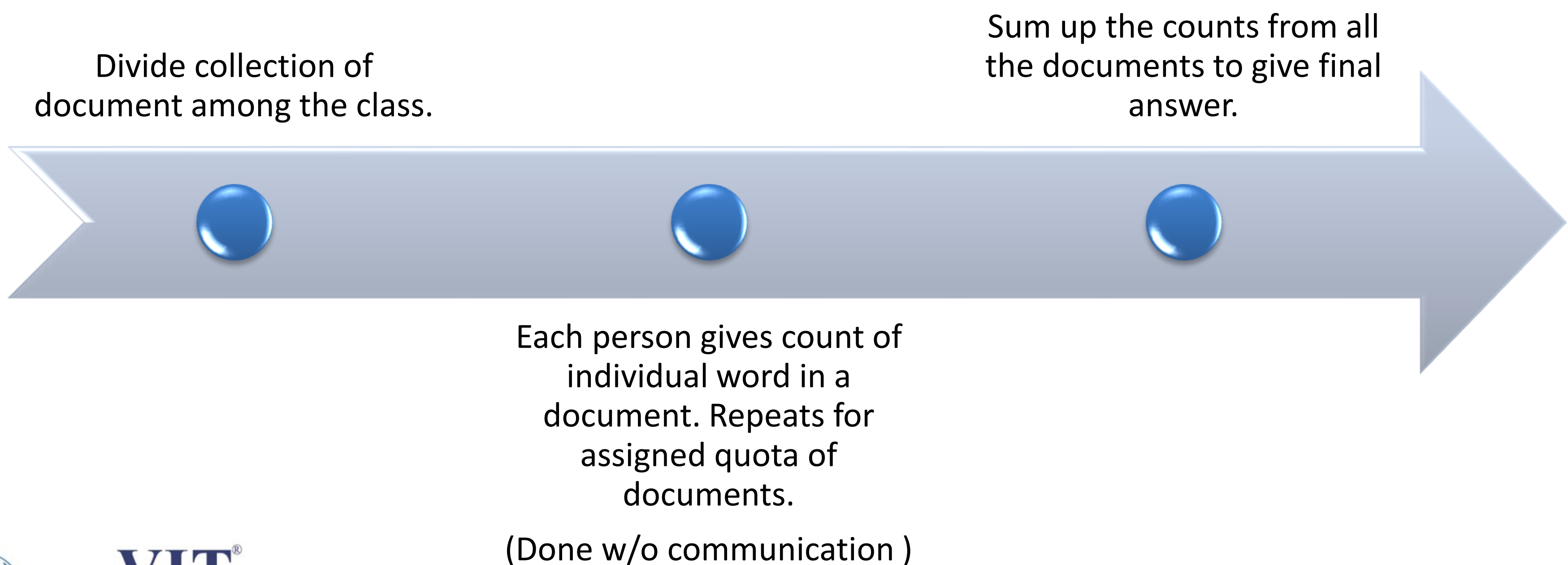
Network Topology In MapReduce



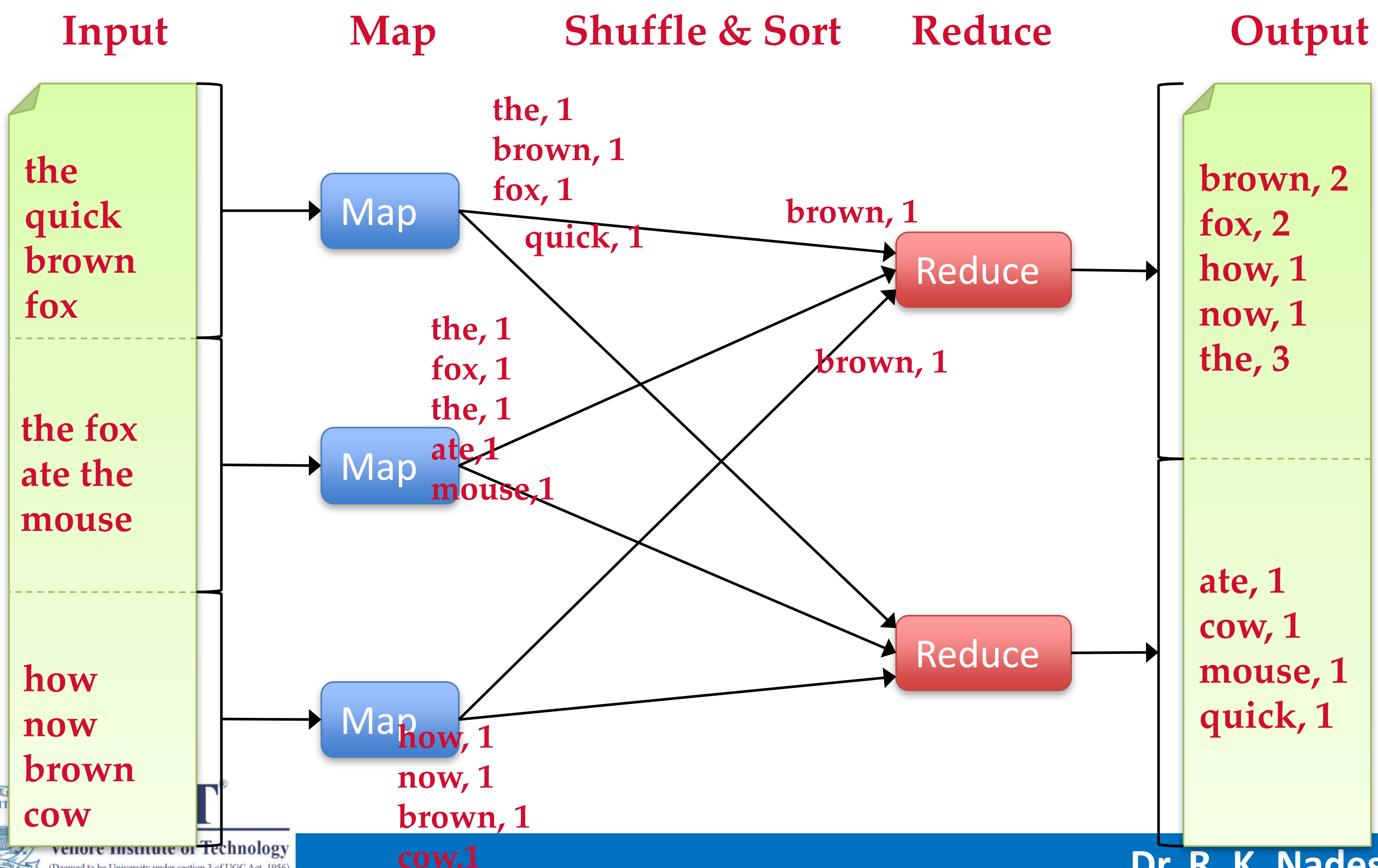
- MapReduce assumes a tree style network topology
- Nodes are spread over different racks embraced in one or many data centers
- A salient point is that the bandwidth between two nodes is dependent on their relative locations in the network topology
- For example, nodes that are on the same rack will have higher bandwidth between them as opposed to nodes that are off-rack

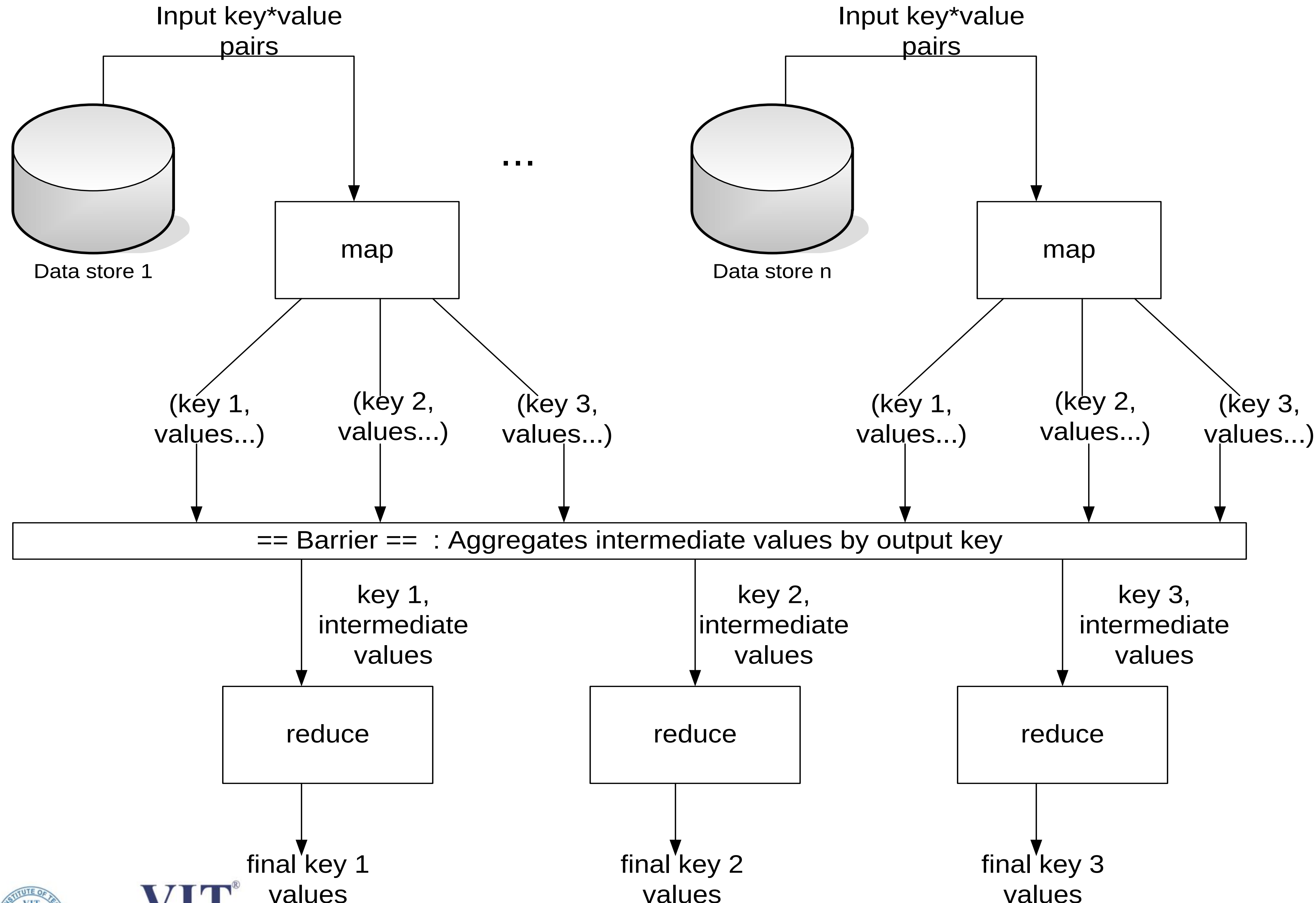
Example: Word counting

Consider the problem of counting the number of occurrences of each word in a large collection of documents”

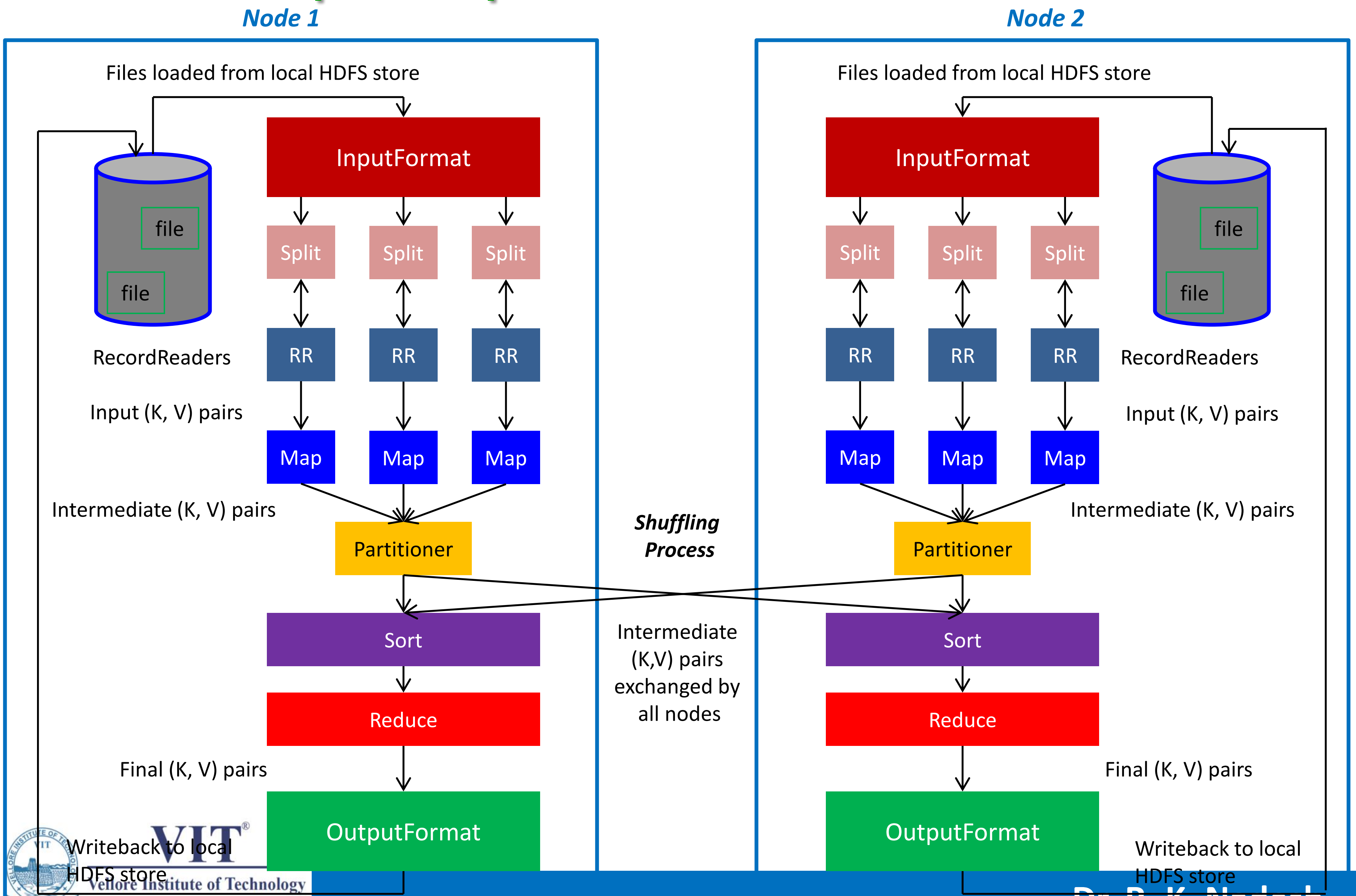


Word Count Execution



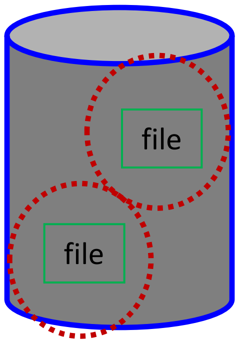

Input key*value pairs
Input key*value pairs
...
Data store 1
map
Data store n
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

Hadoop MapReduce: A Closer Look



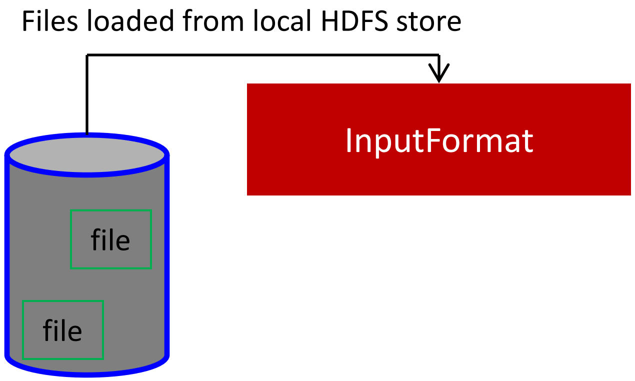
Input Files

- *Input files* are where the data for a MapReduce task is initially stored
- The input files typically reside in a distributed file system (e.g. HDFS)
- The format of input files is arbitrary
  - Line-based log files
  - Binary files
  - Multi-line input records
  - Or something else entirely



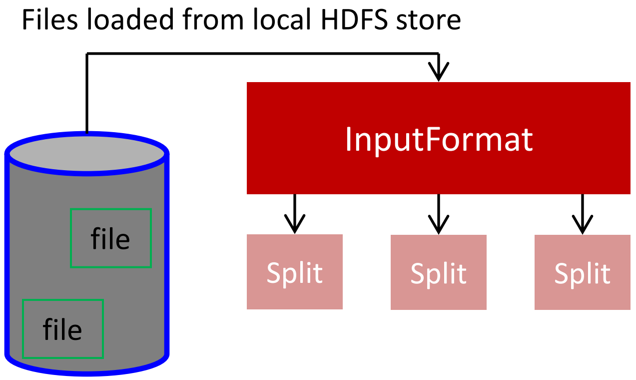
InputFormat

- How the input files are split up and read is defined by the *InputFormat*
- InputFormat is a class that does the following:
  - Selects the files that should be used for input
  - Defines the *InputSplits* that break a file
  - Provides a factory for *RecordReader* objects that read the file



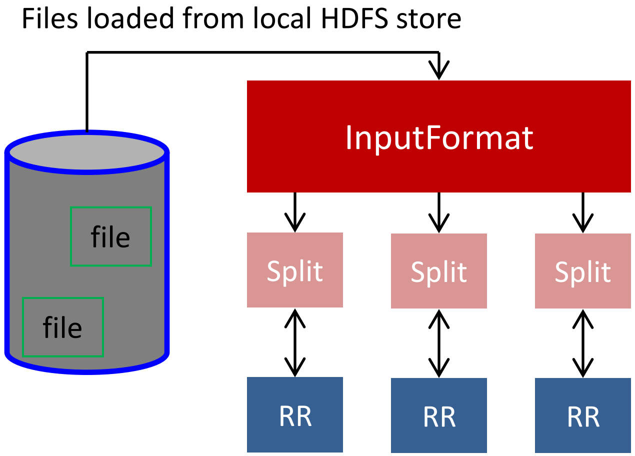
Input Splits

- An *input split* describes a unit of work that comprises a single map task in a MapReduce program
- By default, the InputFormat breaks a file up into 64MB splits
- By dividing the file into splits, we allow several map tasks to operate on a single file in parallel
- If the file is very large, this can improve performance significantly through parallelism
- Each map task corresponds to a *single* input split



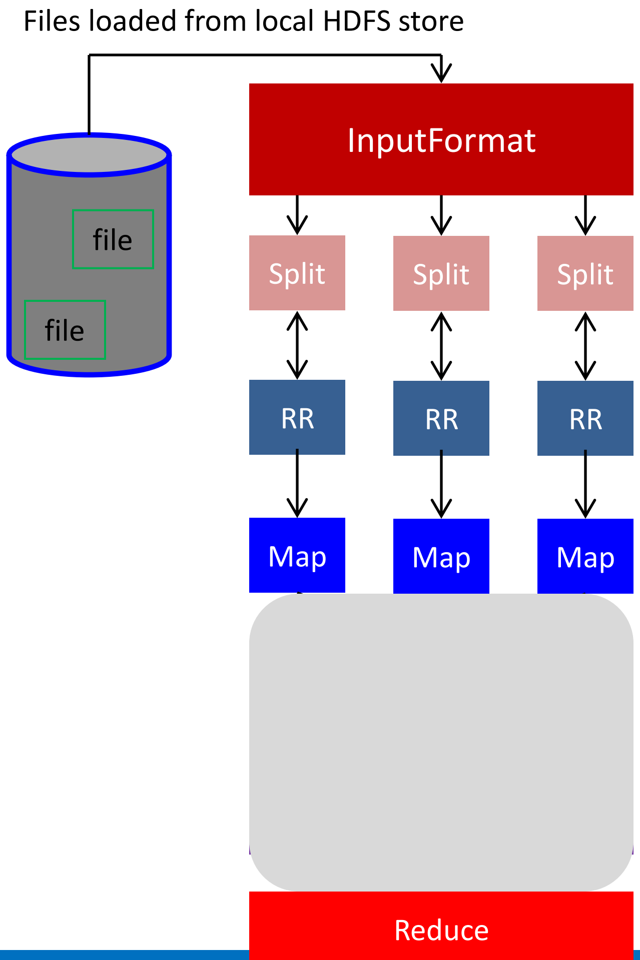
RecordReader

- The input split defines a slice of work but does not describe how to access it
- The *RecordReader* class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers
- The RecordReader is invoked repeatedly on the input until the entire split is consumed
- Each invocation of the RecordReader leads to another call of the map function defined by the programmer



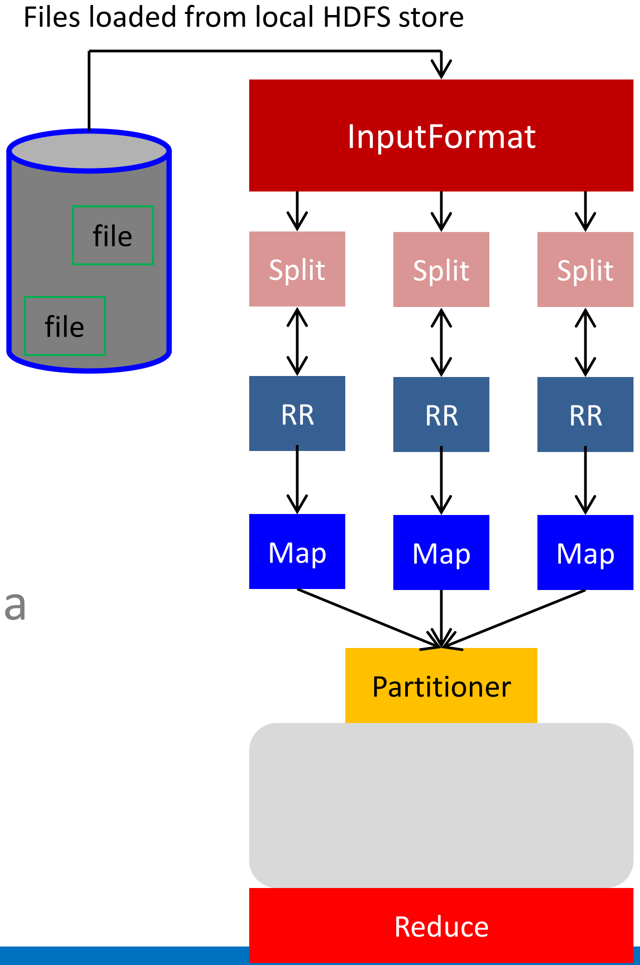
Mapper and Reducer

- The *Mapper* performs the user-defined work of the first phase of the MapReduce program
- A new instance of Mapper is created for each split
- The *Reducer* performs the user-defined work of the second phase of the MapReduce program
- A new instance of Reducer is created for each partition
- *For each key in the partition assigned to a Reducer, the Reducer is called once*



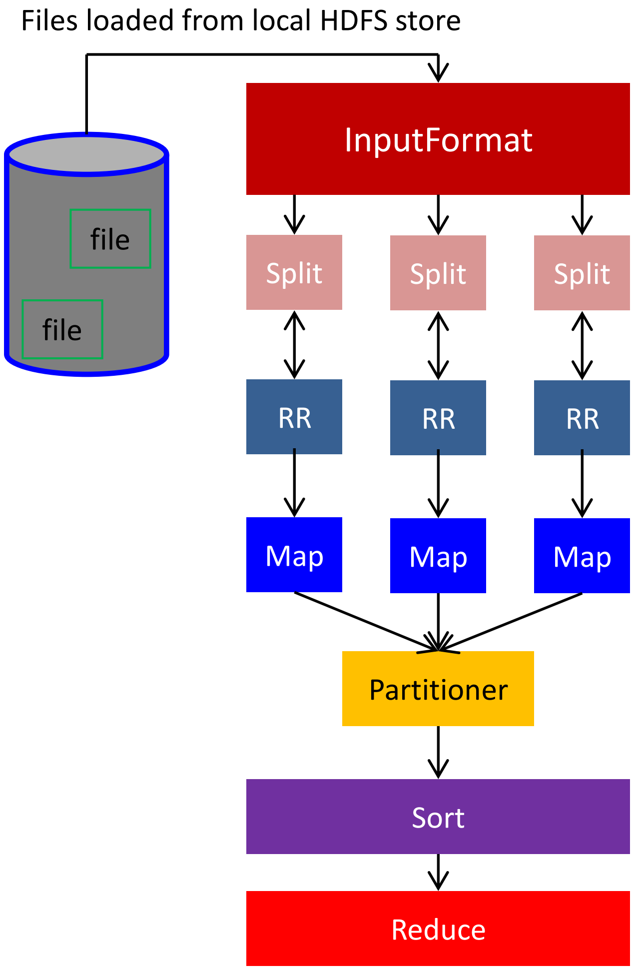
Partitioner

- Each mapper may emit (K, V) pairs to *any* partition
- Therefore, the map nodes must all agree on where to send different pieces of intermediate data
- The ***partitioner*** class determines which partition a given (K,V) pair will go to
- The default partitioner computes ***a hash value*** for a given key and assigns it to a partition based on this result



Sort

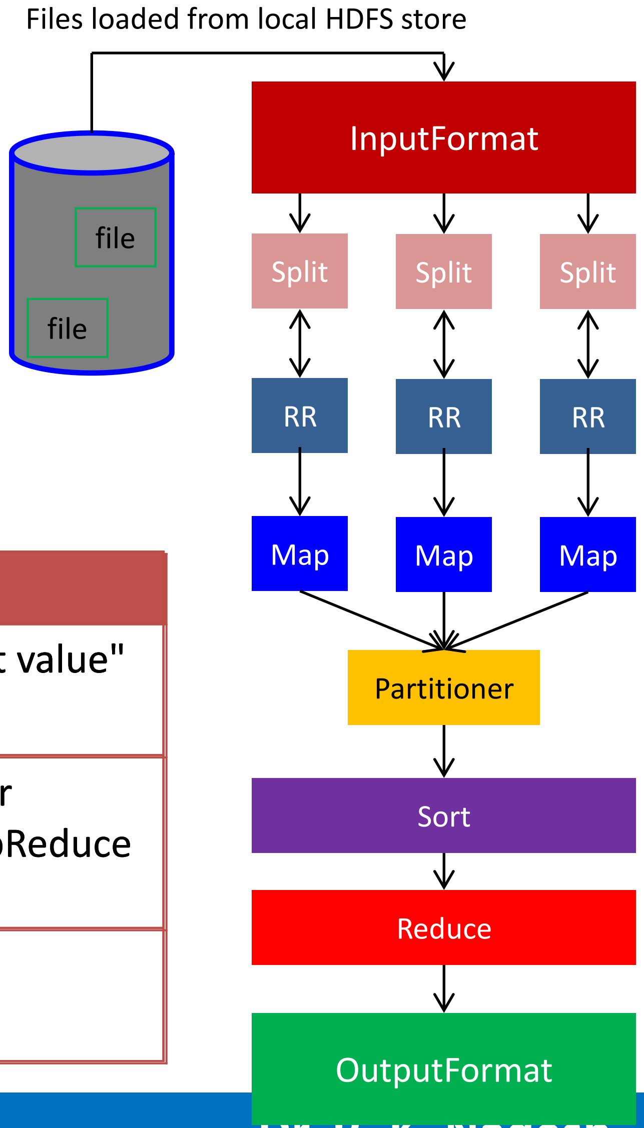
- Each Reducer is responsible for reducing the values associated with (several) intermediate keys
- The set of intermediate keys on a single node is ***automatically sorted*** by MapReduce before they are presented to the Reducer



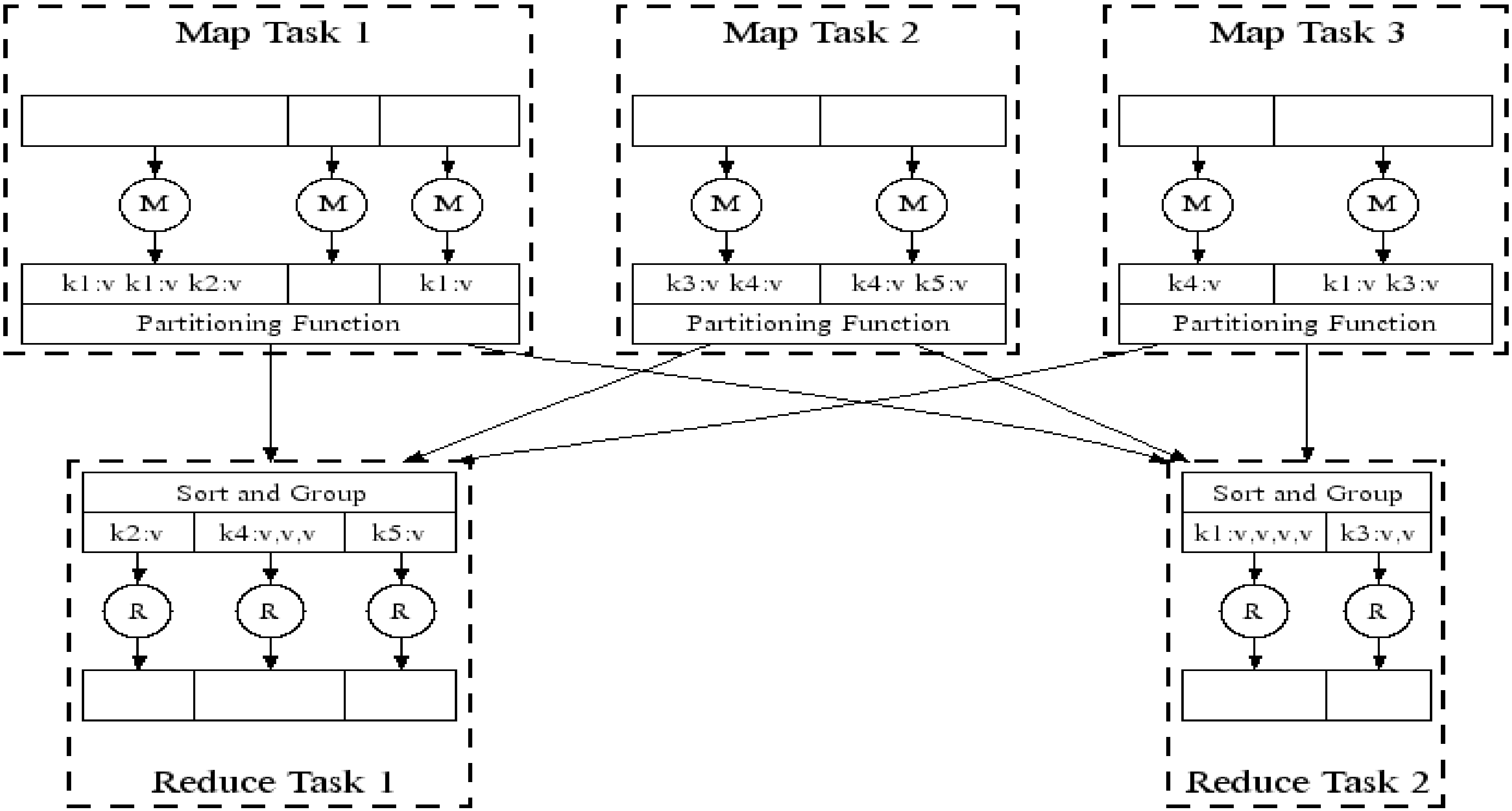
OutputFormat

- The *OutputFormat* class defines the way (K,V) pairs produced by Reducers are written to output files
- The instances of OutputFormat provided by Hadoop write to files on the local disk or in HDFS
- Several OutputFormats are provided by Hadoop:

OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files

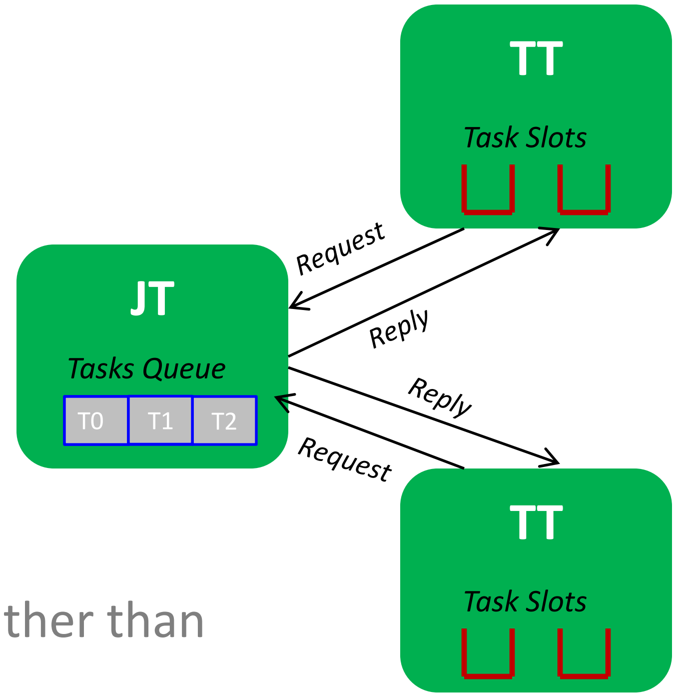


Parallel Efficiency of Map Reduce



Task Scheduling in MapReduce

- MapReduce adopts a *master-slave architecture*
- The master node in Map Reduce is referred to as *Job Tracker* (JT)
- Each slave node in MapReduce is referred to as *Task Tracker* (TT)
- MapReduce adopts a *pull scheduling* strategy rather than a *push one*
  - I.e., JT does not push map and reduce tasks to TTs but rather TTs pull them by making pertaining requests



Map and Reduce Task Scheduling

- Every TT sends a *heartbeat message* periodically to JT encompassing a request for a map or a reduce task to run

I. Map Task Scheduling:

- JT satisfies requests for map tasks via attempting to schedule mappers in the *vicinity* of their input splits (i.e., it considers locality)

II. Reduce Task Scheduling:

- However, JT simply assigns the next yet-to-run reduce task to a requesting TT regardless of TT's network location and its implied effect on the reducer's shuffle time (i.e., it does not consider locality)

Job Scheduling in MapReduce

- In MapReduce, an application is represented as a *job*
- A job encompasses multiple map and reduce tasks
- MapReduce in Hadoop comes with a choice of schedulers:
  - The default is the *FIFO scheduler* which schedules jobs in order of submission
  - There is also a multi-user scheduler called the *Fair scheduler* which aims to give every user a fair share of the cluster capacity over time

Fault Tolerance in Hadoop

- MapReduce can guide jobs toward a successful completion even when jobs are run on a large cluster where probability of failures increases

- The primary way that MapReduce achieves fault tolerance is through *restarting tasks*

- If a TT fails to communicate with JT for a period of time (by default, 1 minute in Hadoop), JT will assume that TT in question has crashed

    - If the job is still in the map phase, JT asks another TT to re-execute *all Mappers that previously ran at the failed TT*

    - If the job is in the reduce phase, JT asks another TT to re-execute *all Reducers that were in progress on the failed TT*

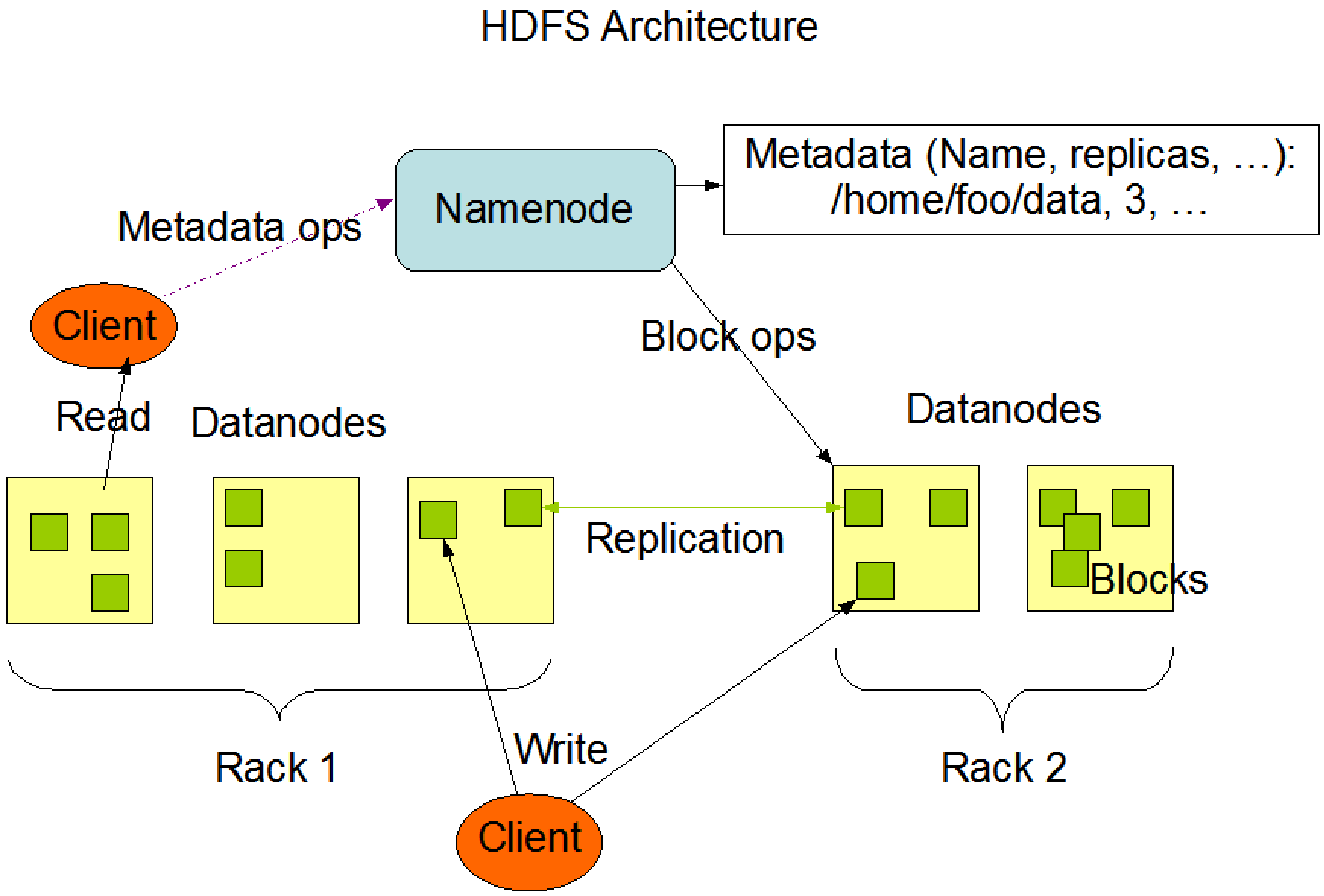
Speculative Execution

- A MapReduce job is dominated by the slowest task
- MapReduce attempts to locate slow tasks (*stragglers*) and run redundant (*speculative*) tasks that will optimistically commit before the corresponding stragglers
- This process is known as *speculative execution*
- Only one copy of a straggler is allowed to be speculated
- Whichever copy (among the two copies) of a task commits first, it becomes the definitive copy, and the other copy is killed by JT

Hadoop Distributed File System(HDFS)

- Very Large Distributed File System
  - 10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Detect failures and recover from them
- Optimized for Batch Processing
  - Data locations exposed so that computations can move to where data resides
  - Provides very high aggregate bandwidth

HDFS Architecture



Functions of a NameNode

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

NameNode Metadata

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- 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
  - Records file creations, file deletions etc

DataNode

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

Block Placement

- Current Strategy
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
  - Additional replicas are randomly placed
- Clients read from nearest replicas
- Would like to make this policy pluggable

Heartbeats

- DataNodes send hearbeat to the NameNode
  - Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure

Replication Engine

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes