Hadoop MapReduce

Anurag Nagar, Ph.D.

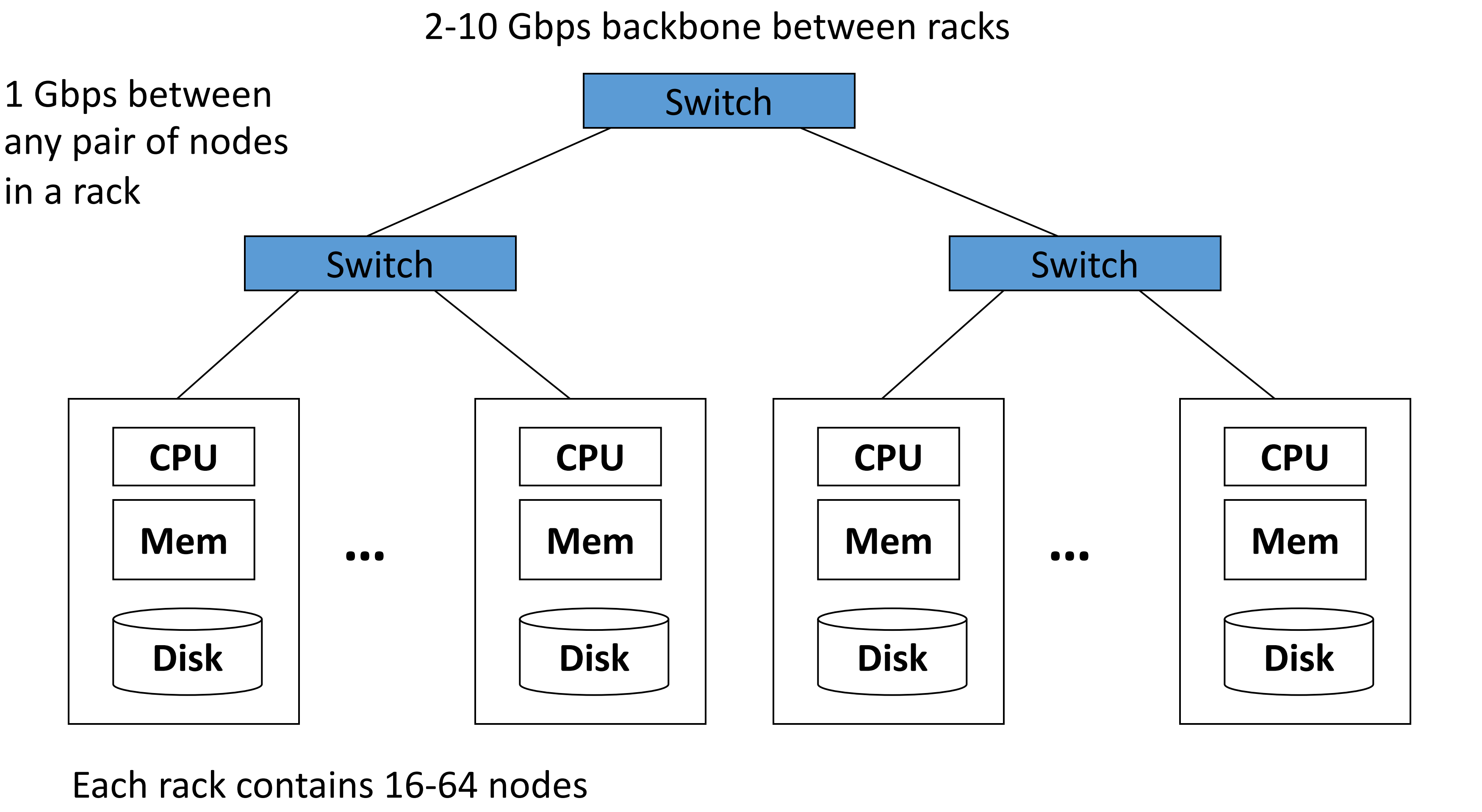
Hadoop Distributed File System (HDFS)

- We learned the architecture of HDFS.
- Each of the data nodes have storage and processing capacity.
- Need a model of data processing that is parallel, distributed, fault-tolerant, and efficient.
- Want to minimize communication between nodes as it costs network bandwidth.
- Let's look at a real example on the next page

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

Cluster Architecture



In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO

Large-scale Computing

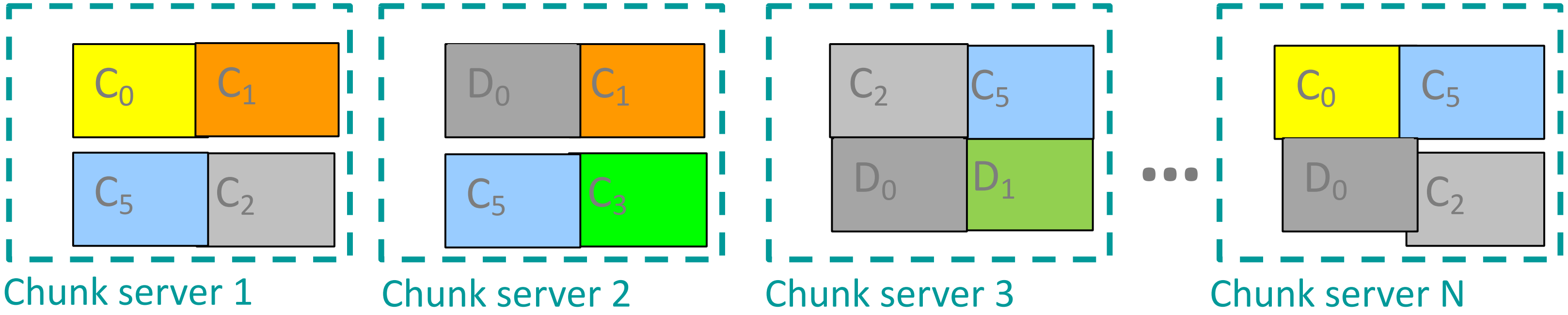
- Problems with using commodity hardware
- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**
  - **Machines fail:**
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to loose 1/day
    - People estimated Google had ~1M machines in 2011
      - 1,000 machines fail every day!

Idea and Solution

- **Issue: Copying data over a network takes time**
- **Idea:**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Map-reduce** addresses these problems
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - Map-Reduce

Distributed File System

- **Reliable distributed file system**
- Data kept in “chunks” spread across machines
- Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

MapReduce in HDFS

- MR is the processing engine of HDFS.
- Helps with the concept of "moving computation" rather than "moving data".
  => locality of computation
- Cluster consists of nodes, that have storage and processing power.
- We need to have multiple nodes perform computation in parallel.

MapReduce

- **Design Considerations:**
  - process vast amounts of data (multi-terabyte data-sets)
  - parallel processing
  - large clusters (thousands of nodes) of commodity hardware
  - reliable
  - fault-tolerant
  - should be able to increase processing power by adding more nodes
    -> "scale-out" and not "scale-up".
  - sharing data or processing between nodes is bad
    -> ideally want "shared-nothing" architecture.
  - want batch processing
    -> process entire dataset and not random seeks

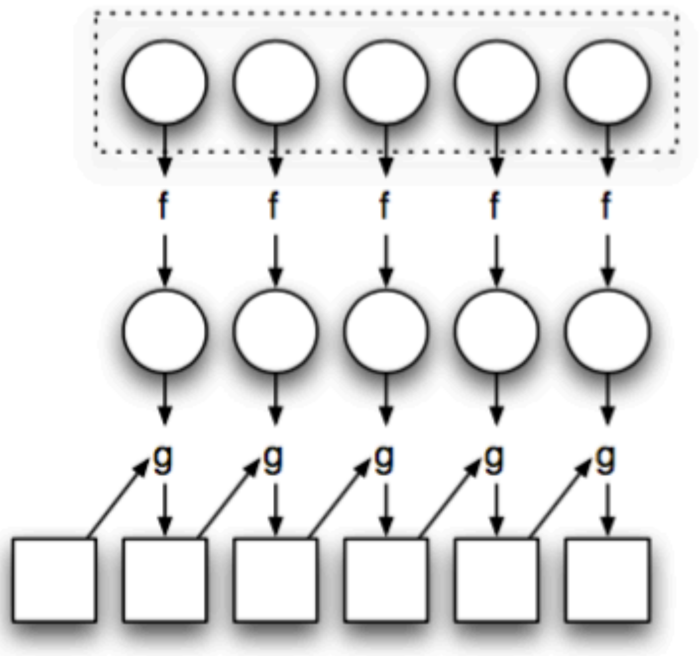
MapReduce Basics

MR has origins in Functional Programming

- Map is a higher order function that applies a function element-wise to a list of elements.
- Map transform **<u>lists</u>** of input data elements into **<u>lists</u>** of output data elements by applying a function to each element of the list.
- Reduce (also called Fold) is a higher order function that processes a list of elements by applying a function pairwise and finally returning a scalar.
- Reduce compacts a list into a scalar by applying a function pairwise.

Functional Programming

- **Key feature: higher order functions**
  - Functions that accept other functions as arguments
  - **Map** and **Fold (Reduce)**



f is applied to every element and it results in a new list

g starts with an initial value and reduces every element
i.e. compacts list to a scalar

Figure: Illustration of *map* and *fold*.

Map Operation

- Define a function:

```
square x = x * x
```

- Apply on a list:

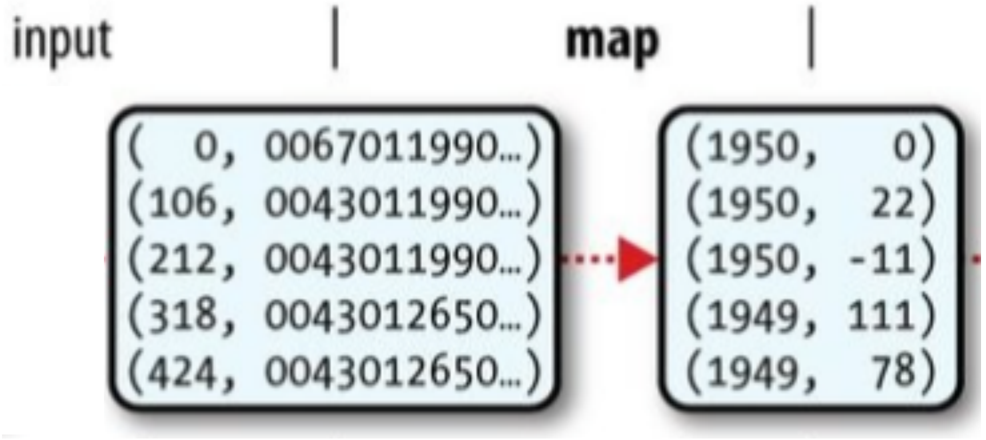
```
>>> map square [1, 2, 3, 4, 5]
```

- Get another list:

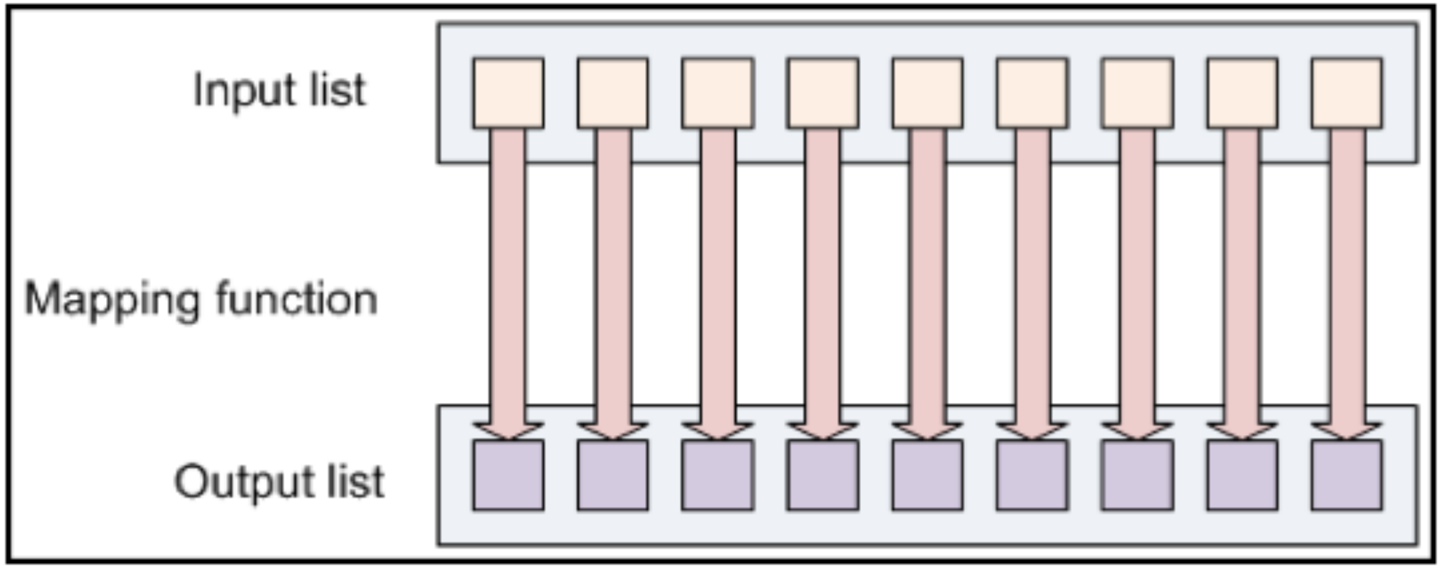
```
[1, 4, 9, 16, 25]
```

Map function

- Takes input (k, v) and outputs (k', v')
  => Generally input k has little meaning, but we try to find a meaningful output k'
- Example: You have input file with line number as key and text as value. A map function could extract and output year as key and temperature as value.



Mapping



Mapper Process

Reduce (Fold) Operation

- Define an operator: +
- Initial value = 0
- Apply on a list: `[1,2,3,4,5]`
- Get a scalar: 15

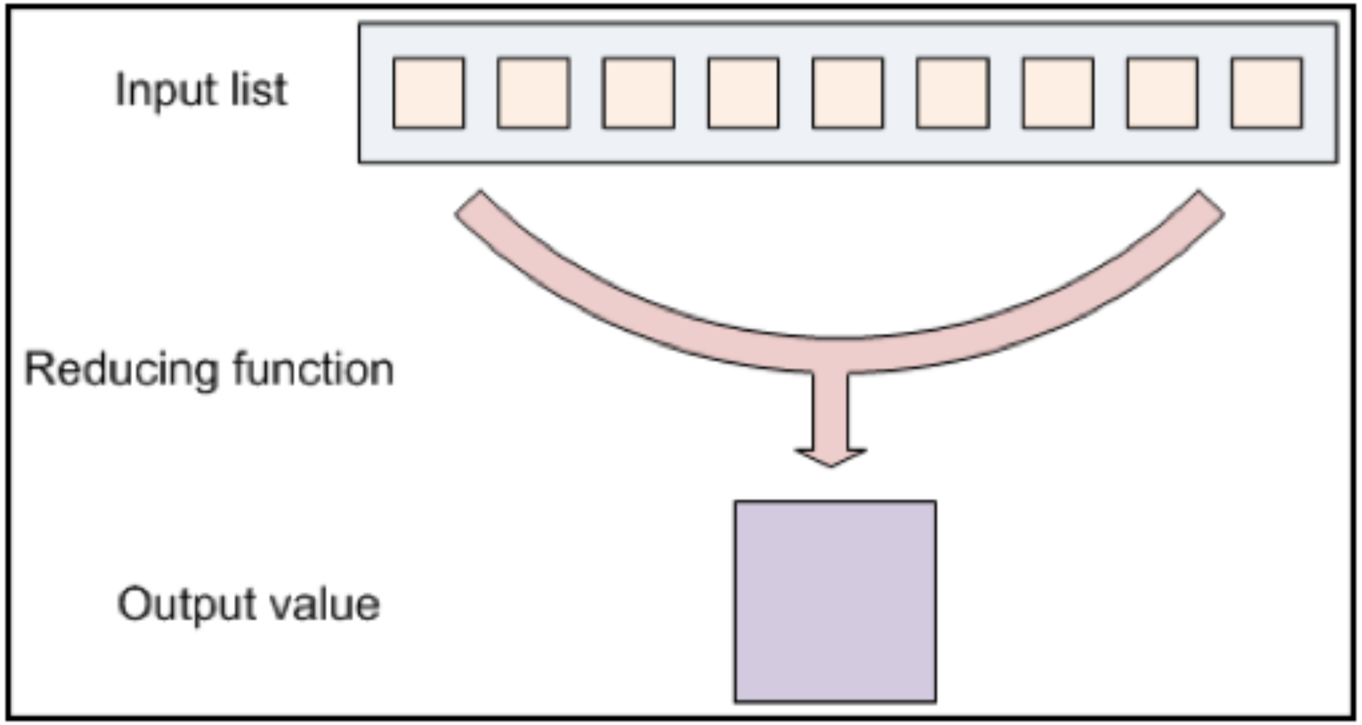
Reduce function

- Reduce function generally receives a key and a list of values.
- It compacts the list to a single (generally) value.
- For example, input key is year and value is list of temperatures. Output could be key and maximum temperature.

  (1950, [30, 70, 50, 72, 18]) -> (1950, 72)

- A key point is that reduce is generally run on data from same key value.
  => Eg. Find average time spent by each visitor on a website
  Key = userID, Value = Time spent during each visit
  It makes sense to aggregate (reduce) for each key separately

Reducing



Reducer Process

MapReduce Data Structures

Key-Value Structure

- Each data element needs to have a **key** associated with it.
- Uniquely identifies the data item.
- Example: Log of cars passing by.
  What's the key?
  Could be the license plate number.

```
AAA-123   65mph, 12:00pm
ZZZ-789   50mph, 12:02pm
AAA-123   40mph, 12:05pm
CCC-456   25mph, 12:15pm
...
```

- Does it have to be unique in entire dataset? No

K-V pairs

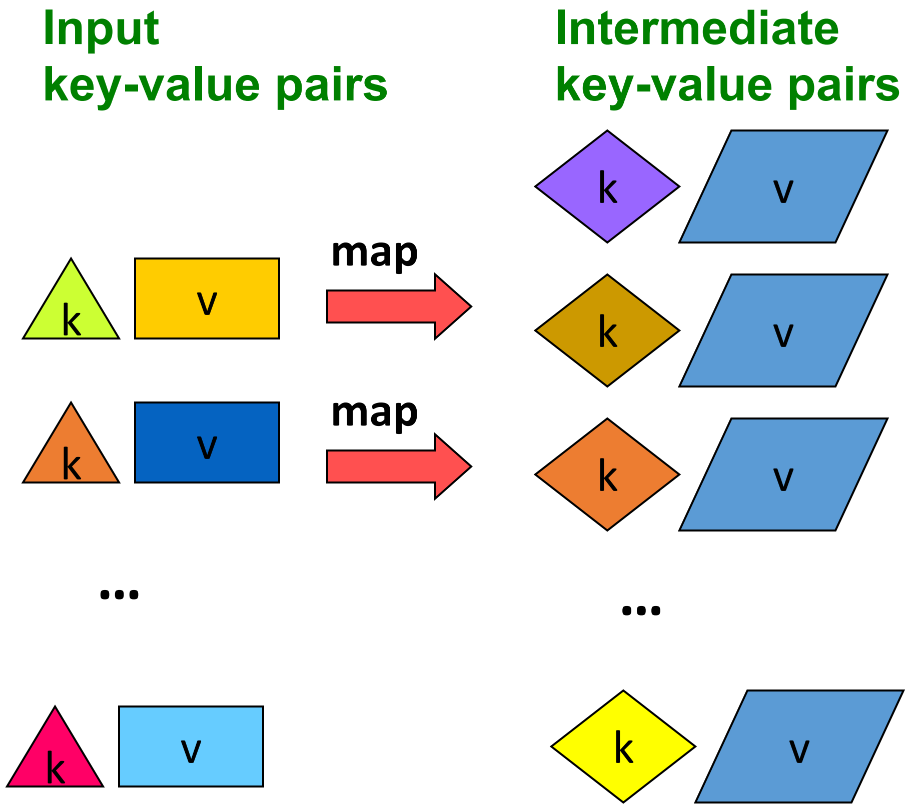
- Key-Value (K-V) pairs are one of the basic data structures for BD.
- Please keep this in mind for future discussion also.

K-V pairs

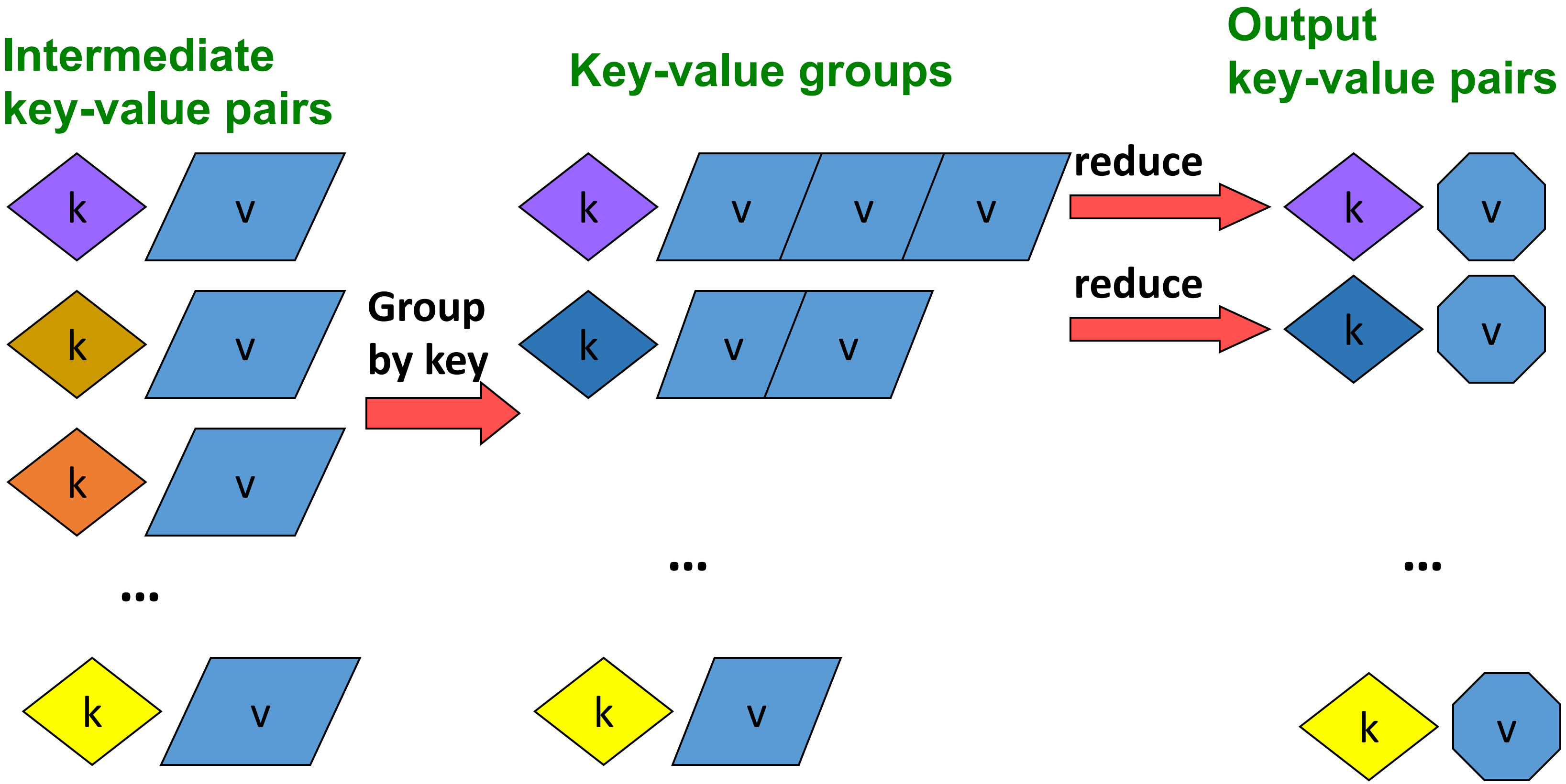
- A mapper is presented data that contains multiple keys.
- It transforms this data in a 1-1 fashion and outputs a meaningful K-V pair.
- The reducer is presented with data containing only a single key.
- It compacts (or aggregates) the values of the key.
- How does each reducer get data from only one key?
- Someone has to do the sorting and shuffling of data from mappers to reducers.
- That's the job of the Hadoop framework

MapReduce in Hadoop

MapReduce: The Map Step



MapReduce: The Reduce Step



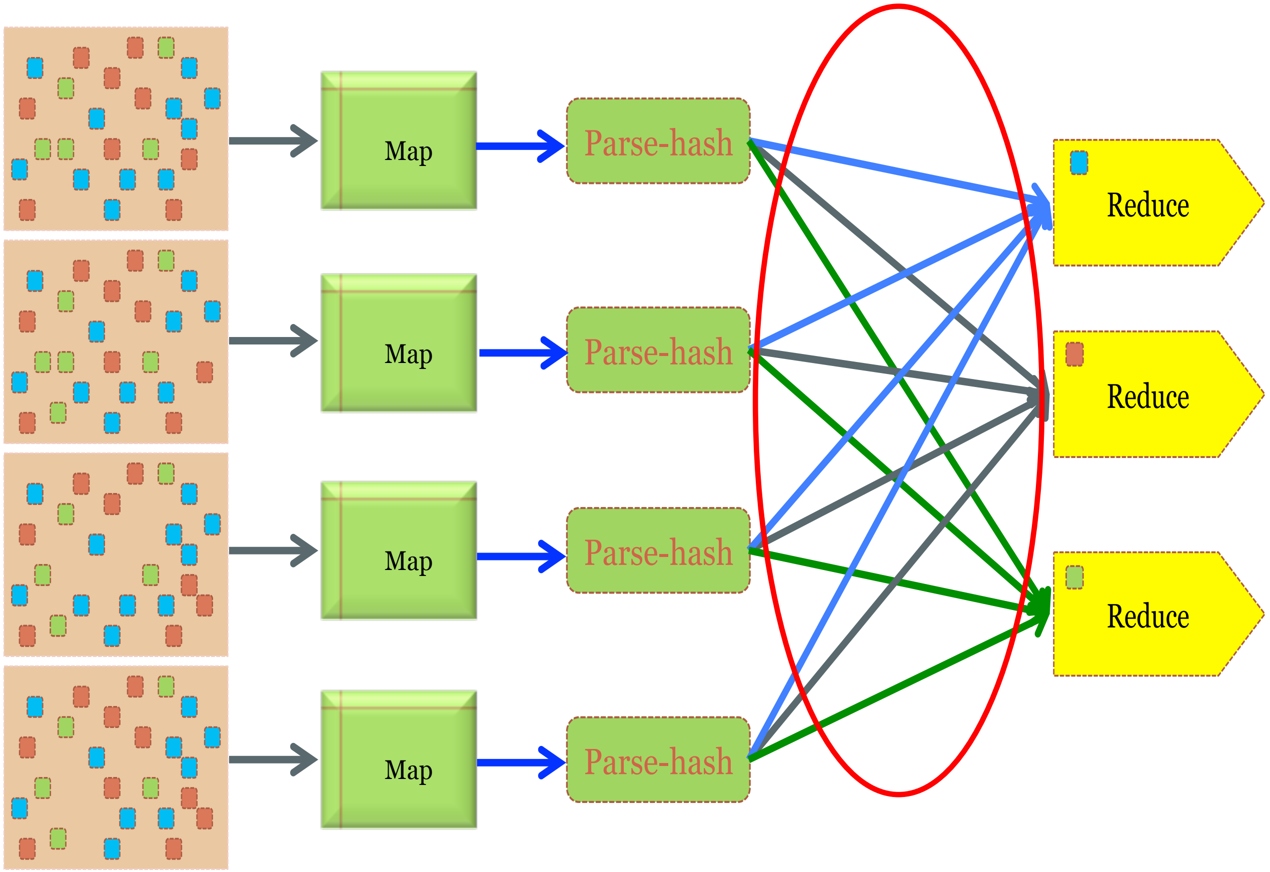
Key-Value Pairs

- Mappers and Reducers are users' code (provided functions)
- Just need to obey the Key-Value pairs interface
- **Mappers:**
  - Consume <key, value> pairs
  - Produce <key, value> pairs
- **Reducers:**
  - Consume <key, <list of values>>
  - Produce <key, value>
- **Shuffling and Sorting:**
  - Hidden phase between mappers and reducers
  - Groups all similar keys from all mappers, sorts and passes them to a certain reducer in the form of <key, <list of values>>

Example 1 – Color Count

MapReduce Execution in Hadoop

- Suppose you are given a dataset where each item is keyed with a color – Red, Blue, or Green
- Aim is to compute the count of each colors.



Dataset is divided into 4 blocks.

The map-reduce job consists of 4 map tasks and 3 reduce tasks

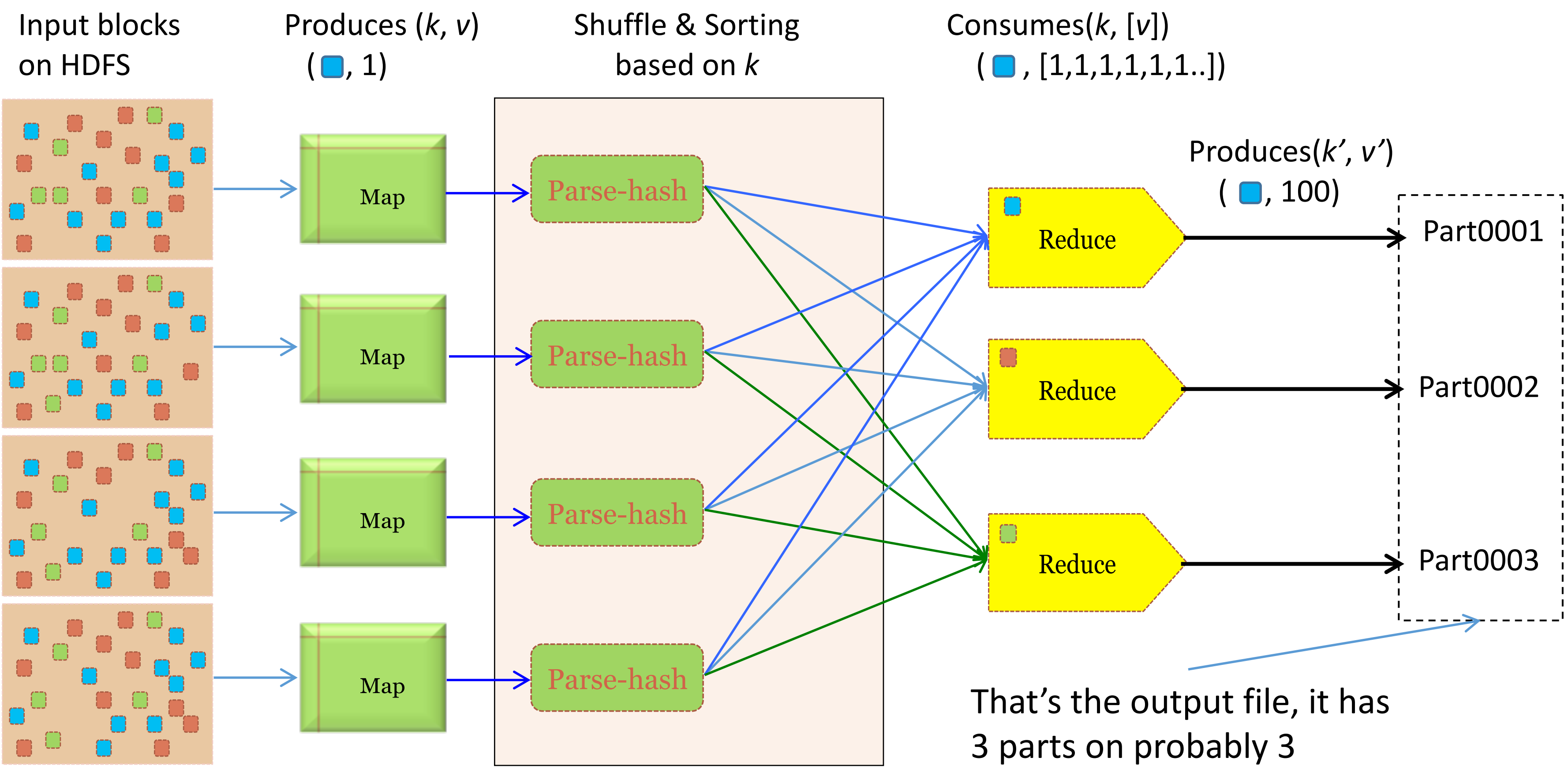
Map task takes each data item and applies a transformation to it. Could be as simple as output (key, 1) e.g. (Red, 1)

Reduce task needs to get data of a single key.

Framework does the sorting and shuffling

Color Count Example

Job: Count the number of each color in a data set



That's the output file, it has 3 parts on probably 3 different machines

Example 2 – Word Count

Programming Model: MapReduce

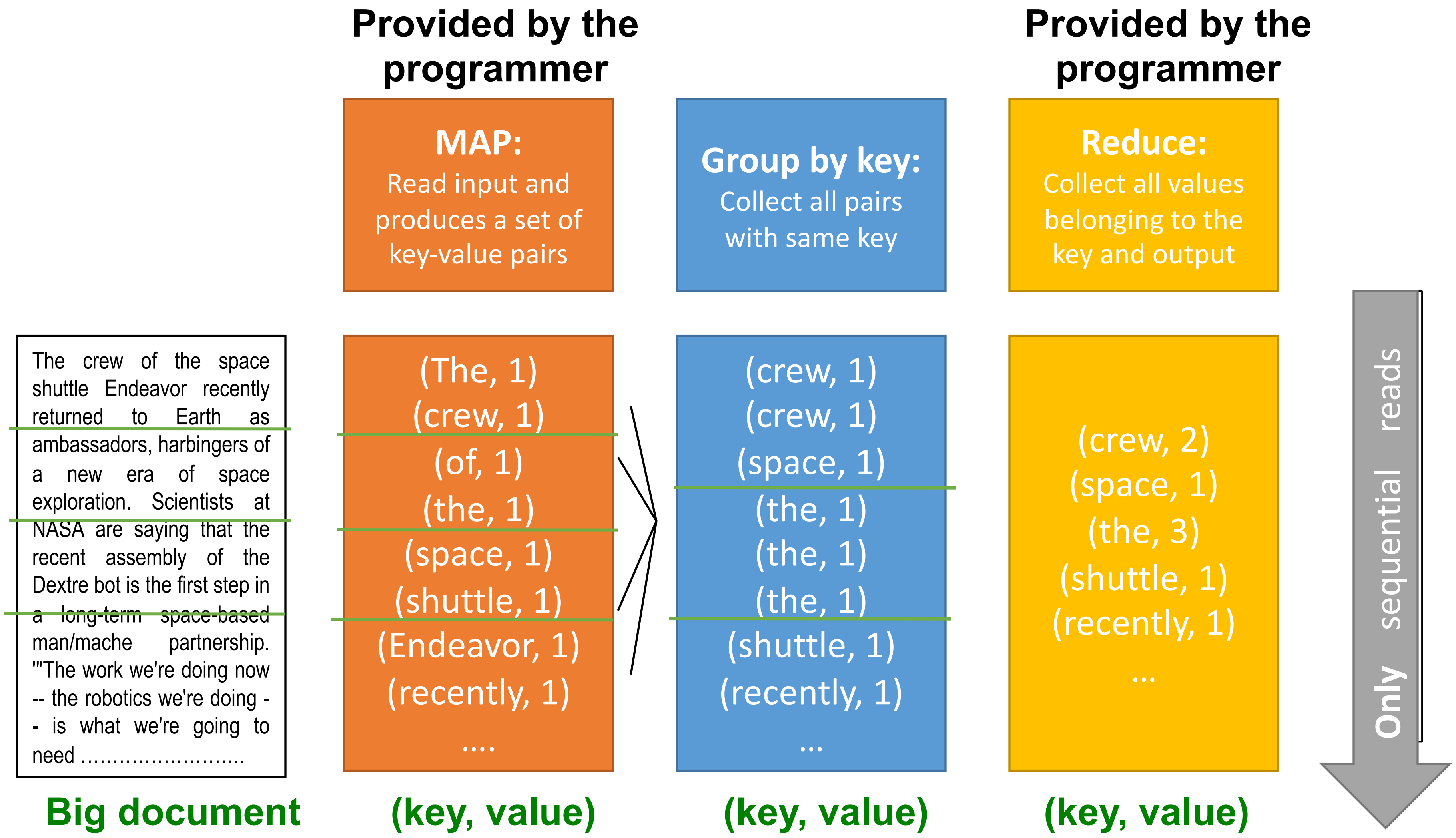
Warm-up task:

- We have a huge text document

- Count the number of times each distinct word appears in the file

- **Sample application:**
  - Analyze web server logs to find popular URLs

MapReduce: Word Counting

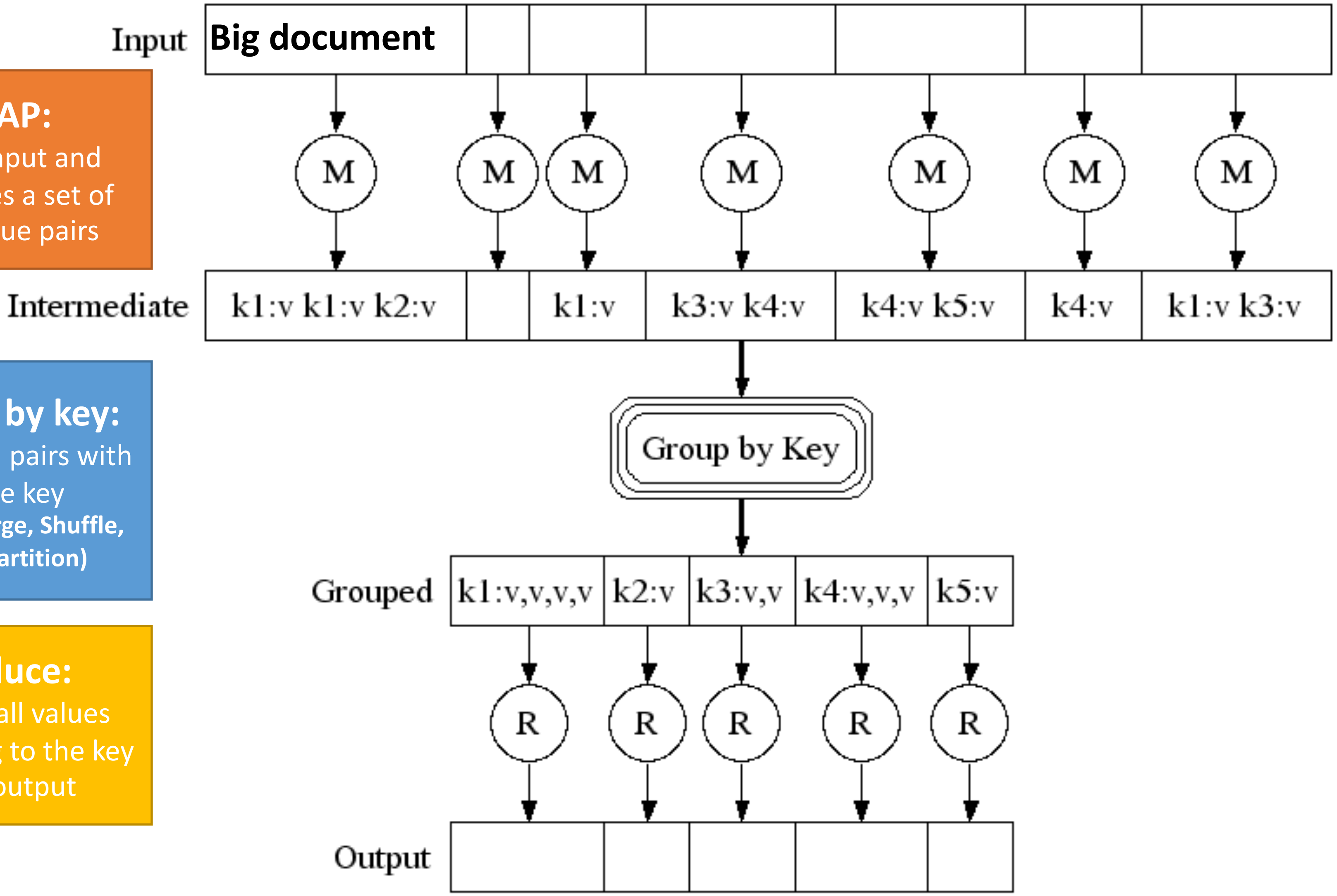


Map-Reduce: A diagram

MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



Word Count Using MapReduce

```
map(key, value):
// key: document name; value: text of the document
  for each word w in value:
      emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

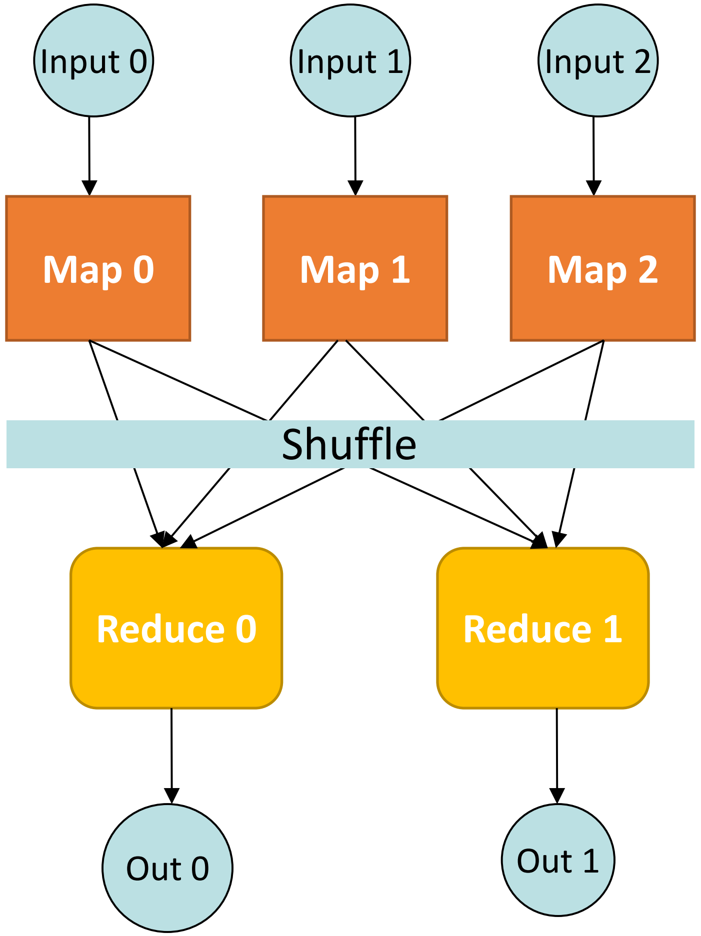
Map-Reduce: Environment

Map-Reduce environment takes care of:

- Partitioning the input data (input splits)
- Scheduling the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine failures
- Managing required inter-machine communication

Map-Reduce

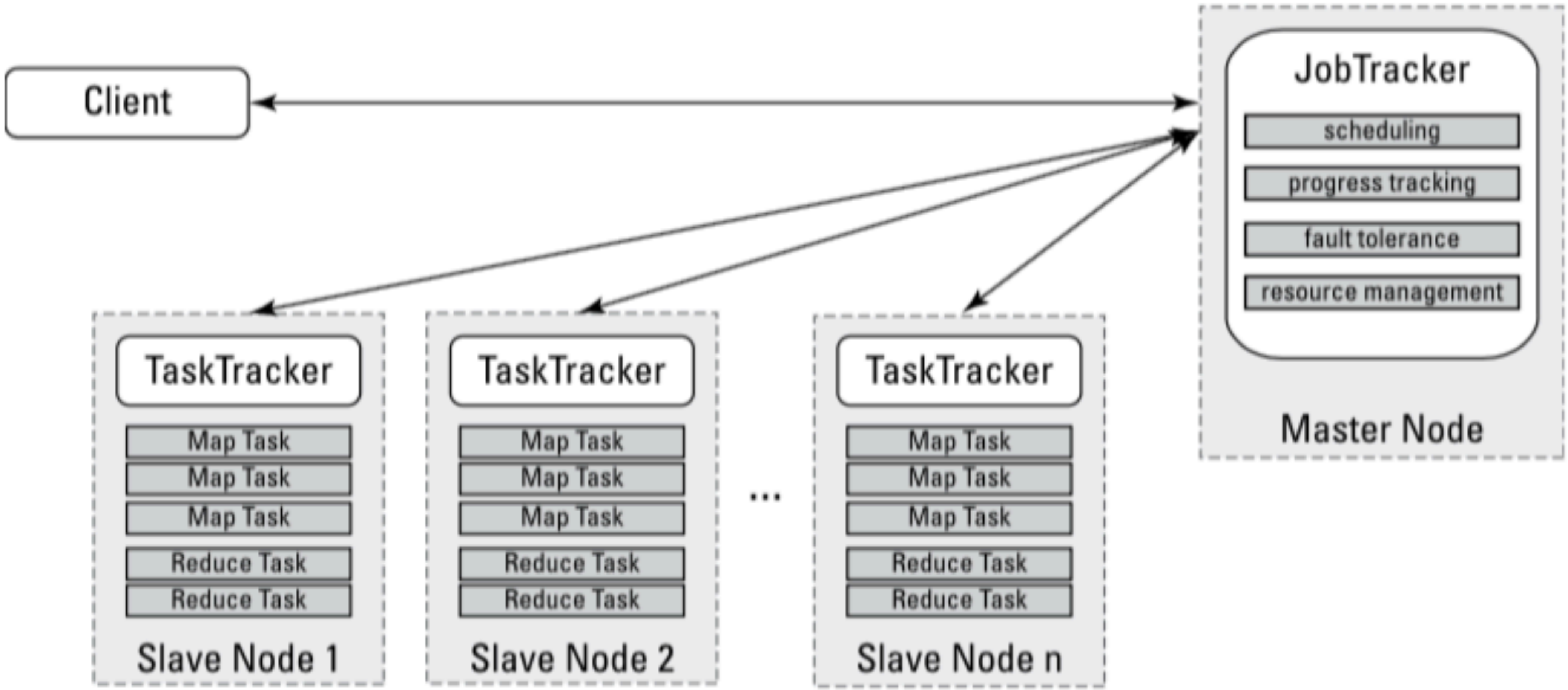
- Programmer specifies:
  - Map and Reduce and input files
- **Workflow:**
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k’v’-pairs with a given k’ are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grouped by key into new k''v''-pairs
  - Write the resulting pairs to files



- All phases are distributed with many tasks doing the work

Hadoop MapReduce Architecture

Trackers


Client
JobTracker
scheduling
progress tracking
fault tolerance
resource management
Master Node
TaskTracker
Map Task
Map Task
Map Task
Reduce Task
Reduce Task
Slave Node 1
TaskTracker
Map Task
Map Task
Map Task
Reduce Task
Reduce Task
Slave Node 2
...
TaskTracker
Map Task
Map Task
Map Task
Reduce Task
Reduce Task
Slave Node n

Isolated Tasks

- Map and Reduce tasks work in isolation from each other
- This is called task isolation.
- Saves bandwidth, no waiting for other nodes.
- Ideally, each node works on local data
  => Idea of moving computation to data
- There is a process called TaskTracker that runs on each DataNode.
- It monitors the tasks and communicates results with a JobTracker that runs on NameNode

Storage

- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data

- **Intermediate results are stored on local FS of Map and Reduce workers**

- **Output is often input to another MapReduce task**

Coordination: Master

- **Master node takes care of coordination:**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer
  - Master pushes this info to reducers

- Master pings workers periodically to detect failures

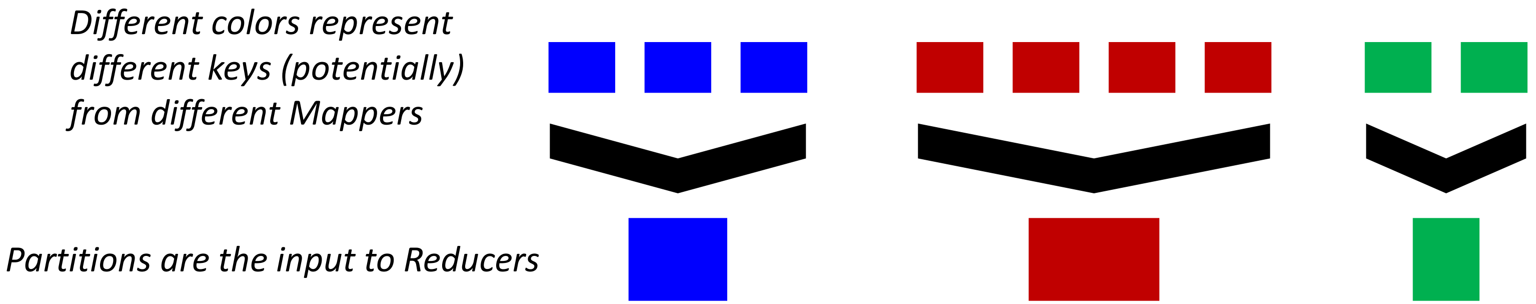
Partitions

- In MapReduce, intermediate output values consist of different keys. Remember, a reduce task works on values with the same key.

- *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*



Refinement: Partition Function

- **Want to control how keys get partitioned**
  - Need to ensure that records with the same intermediate key end up at the same reducer.
- **System uses a default partition function:**
  - **hash(key) mod *R***
- **Sometimes useful to override the hash function:**
  - E.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

Task and Job Scheduling in MapReduce

Task Scheduling in MapReduce

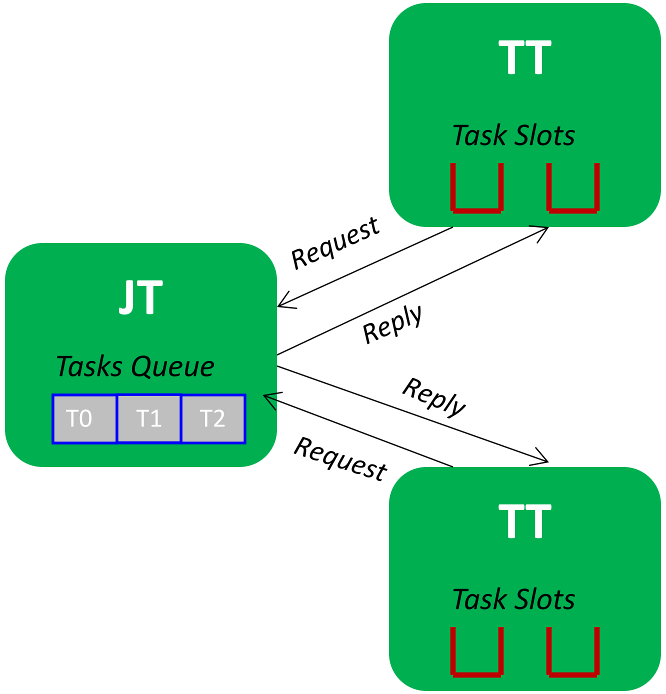
- MapReduce adopts a *master-slave architecture*

- The master node in MapReduce 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:

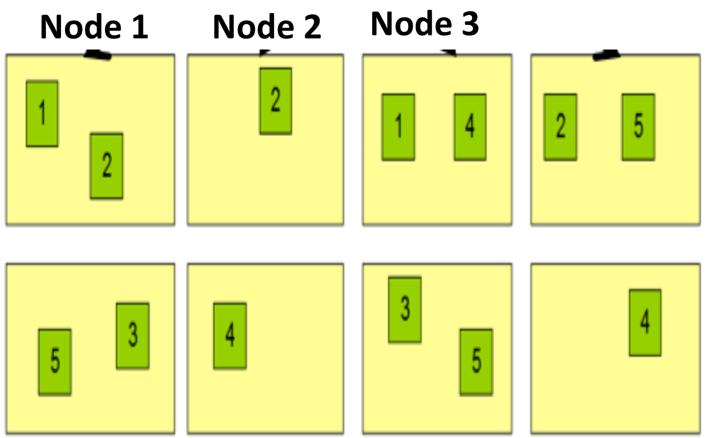
  - However, JT simply assigns the next yet-to-run reduce task to a requesting TT regardless of TT's network location and its implied effect on the reducer's shuffle time (i.e., it does not consider locality)

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

Properties of MapReduce Engine

- **Job Tracker is the master node (runs with the namenode)**
  - Receives the user's job
  - Decides on how many tasks will run (number of mappers)
  - Decides on where to run each mapper (concept of locality)



- This file has 5 Blocks → run 5 map tasks
- Where to run the task reading block "1"
  - *Try to run it on Node 1 or Node 3*

How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

Dealing with Failures

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

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

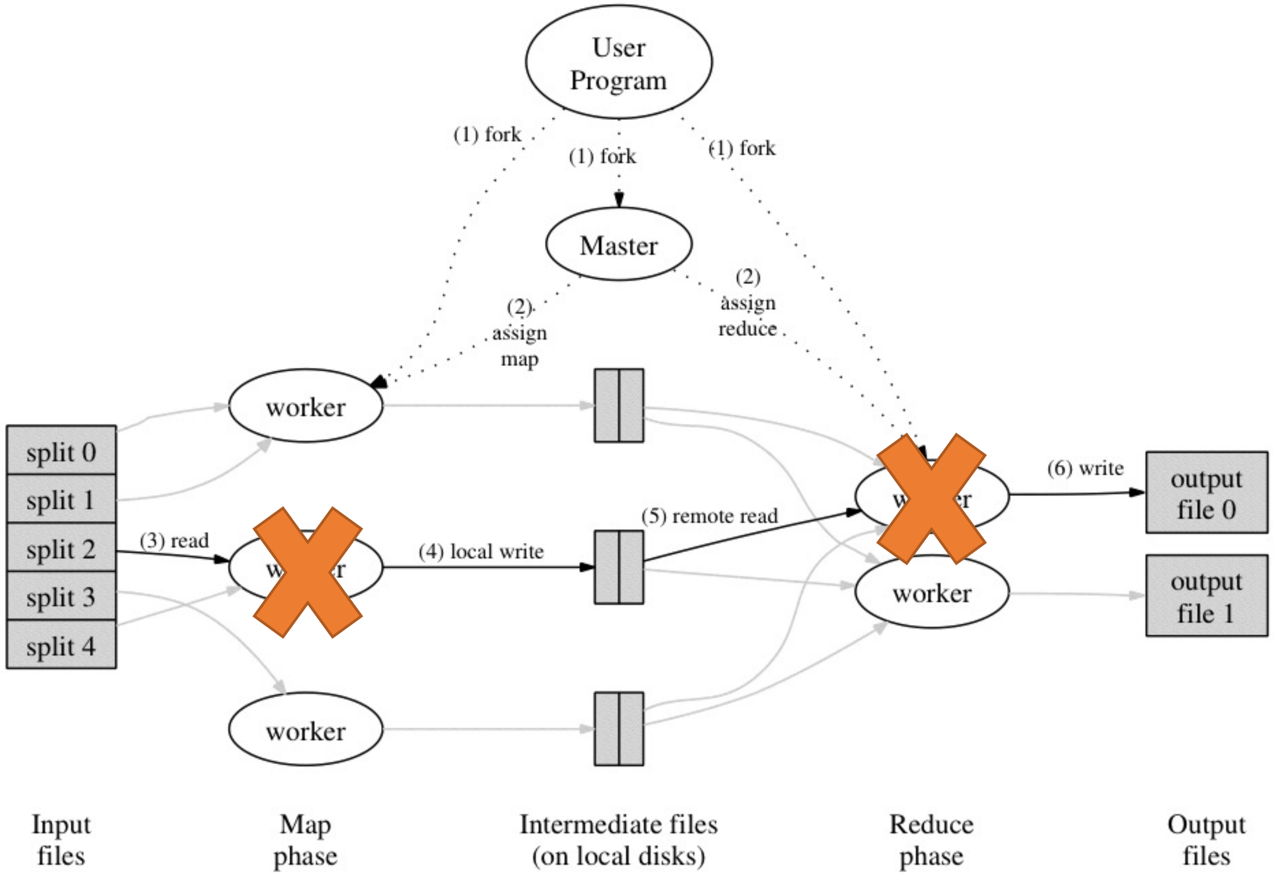
- The primary way that MapReduce achieves fault tolerance is through *restarting tasks*

- 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

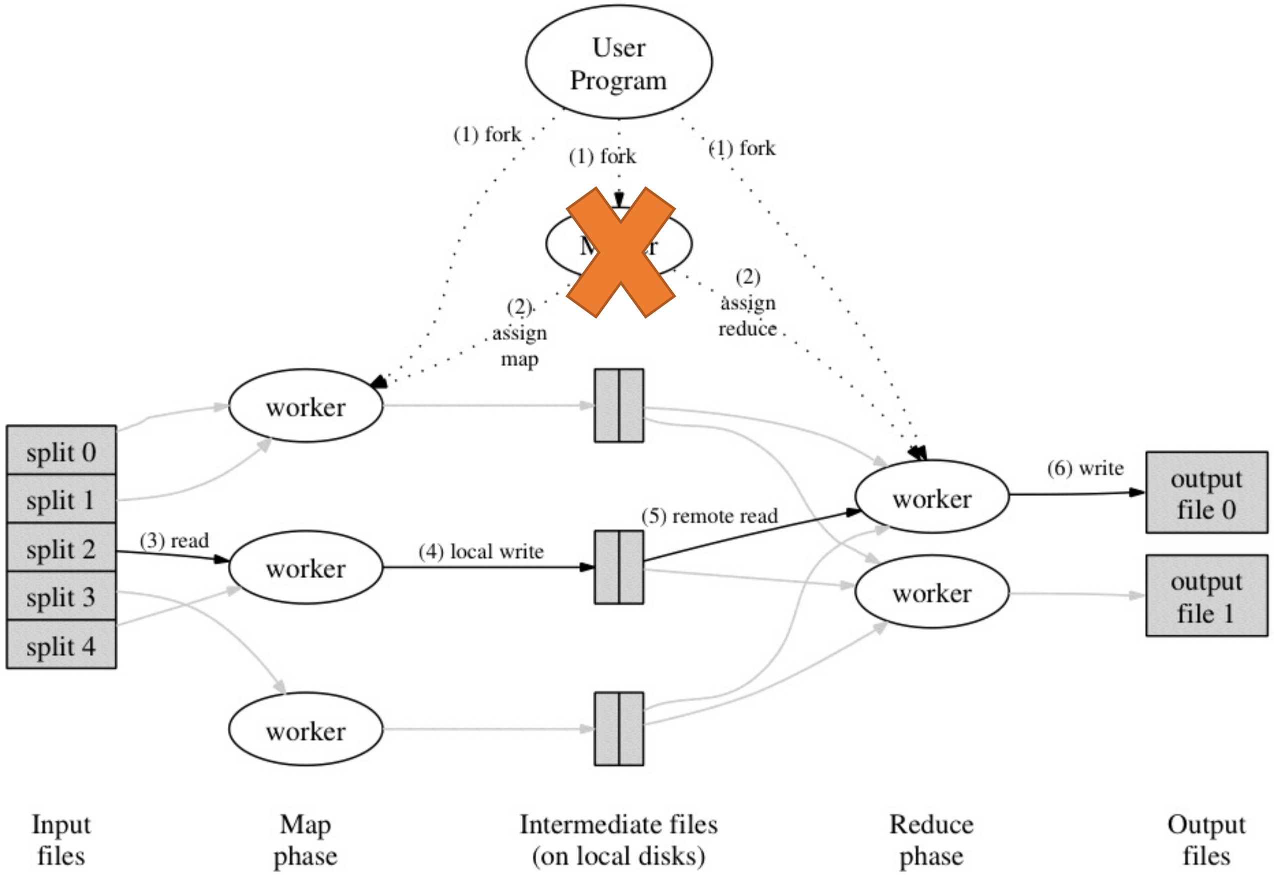
  - If the job is still in the map phase, JT asks another TT to re-execute *all Mappers that previously ran at the failed TT*

  - 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*

Worker Failure



Master Failure



Fault Tolerance / Workers

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed + in-progress *map* tasks
    - Why? Because their output are stored on local disk, and therefore not accessible.
- Re-execute in progress *reduce* tasks
- Task completion committed through master

Robust:

lost 1600/1800 machines once → finished ok

Semantics in presence of failures: see paper

Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”
- **Effect**
  - Dramatically shortens job completion time

Speculative Execution

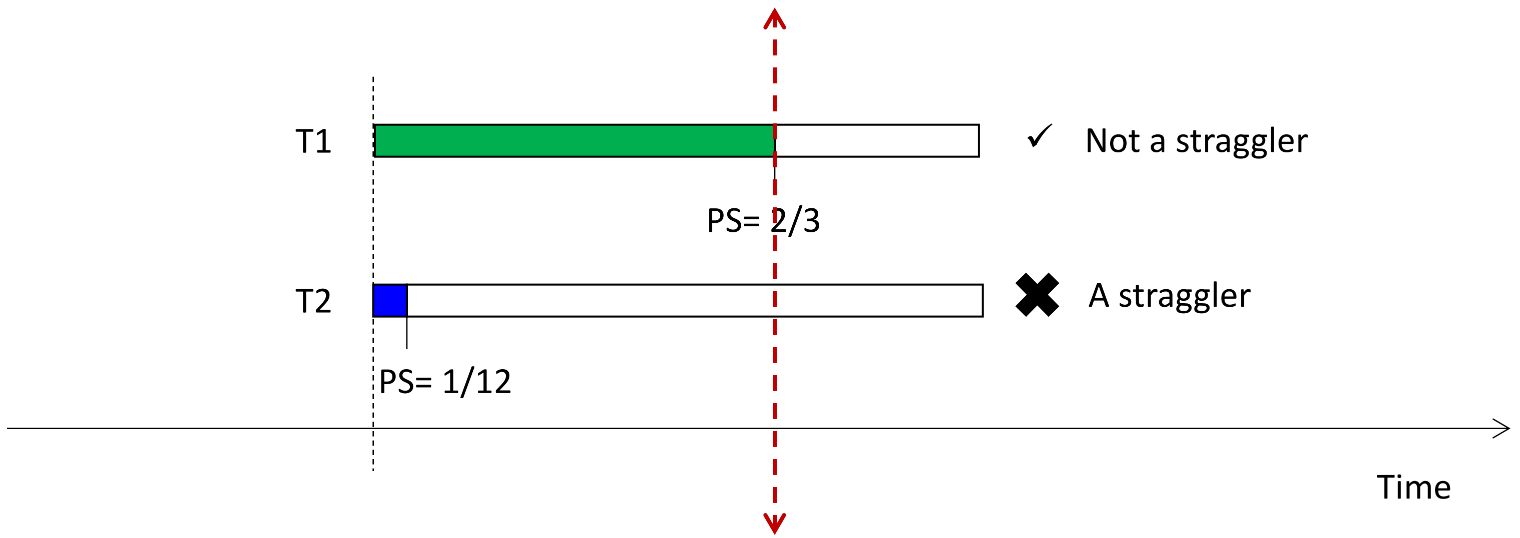
- A MapReduce job is dominated by the slowest task
- MapReduce attempts to locate slow tasks (*stragglers*) and run redundant (*speculative*) tasks that will optimistically commit before the corresponding stragglers
- This process is known as *speculative execution*
- Only one copy of a straggler is 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

Locating Stragglers

- How does Hadoop locate stragglers?

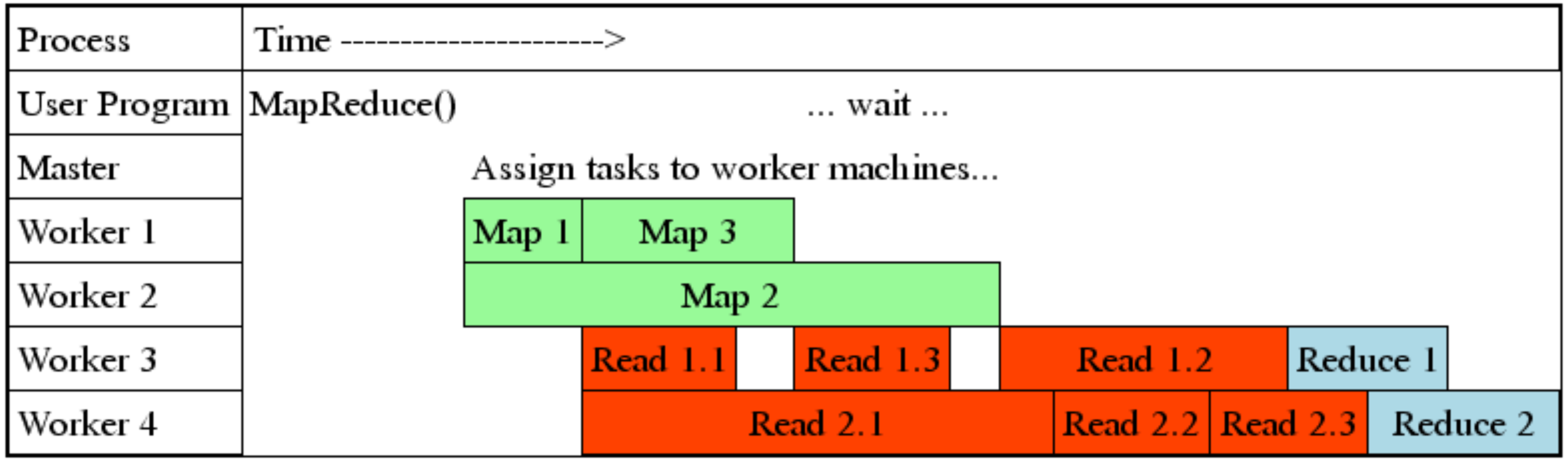
    - Hadoop monitors each task progress using a *progress score* between 0 and 1

    - If a task's progress score ***is less than*** (average – 0.2), and the task has run for at least 1 minute, it is marked as a straggler



Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing



Bigger Picture: Hadoop vs. Other Systems

	Distributed Databases	Hadoop
Computing Model	- Notion of transactions - Transaction is the unit of work - ACID properties, Concurrency control	- Notion of jobs - Job is the unit of work - No concurrency control
Data Model	- Structured data with known schema - Read/Write mode	- Any data will fit in any format - (un)(semi)structured - ReadOnly mode
Cost Model	- Expensive servers	- Cheap commodity machines
Fault Tolerance	- Failures are rare - Recovery mechanisms	- Failures are common over thousands of machines - Simple yet efficient fault tolerance
Key Characteristics	- Efficiency, optimizations, fine-tuning	- Scalability, flexibility, fault tolerance

- ***Cloud Computing***
  - A computing model where any computing infrastructure can run on the cloud
  - Hardware & Software are provided as remote services
  - Elastic: grows and shrinks based on the user's demand
  - Example: Amazon EC2