CS60021: Scalable Data Mining

Map Reduce

Sourangshu Bhattacharya

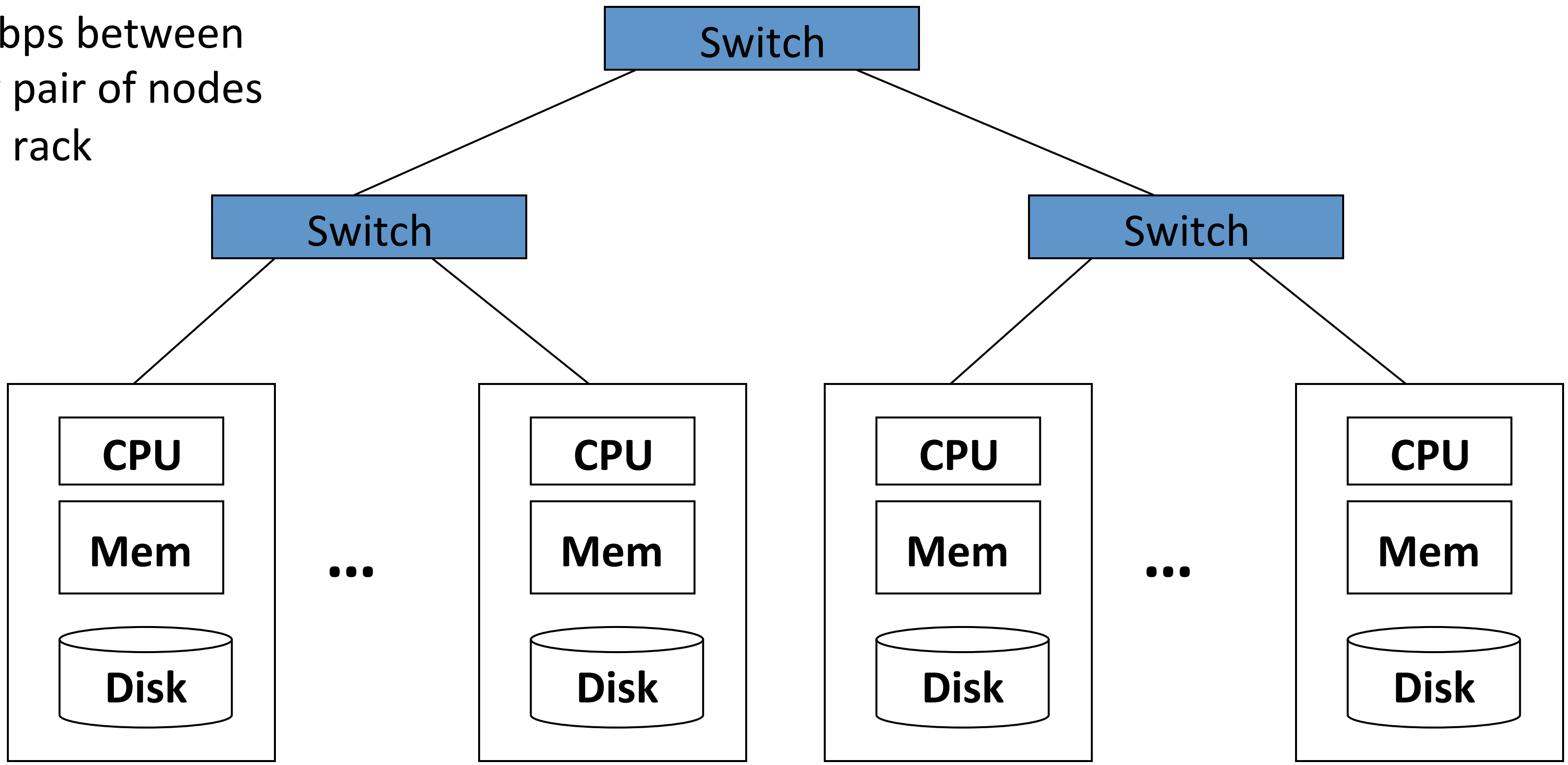
Motivation: Google Example

- 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

2-10 Gbps backbone between racks

1 Gbps between any pair of nodes in a rack



Each rack contains 16-64 nodes

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

Large-scale Computing

- **Large-scale computing** for **data mining** problems on **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!

Big Data Challenges

- Scalability: processing should scale with increase in data.
- Fault Tolerance: function in presence of hardware failure
- Cost Effective: should run on commodity hardware
- Ease of use: programs should be small
- Flexibility: able to process unstructured data

- Solution: Map Reduce !

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

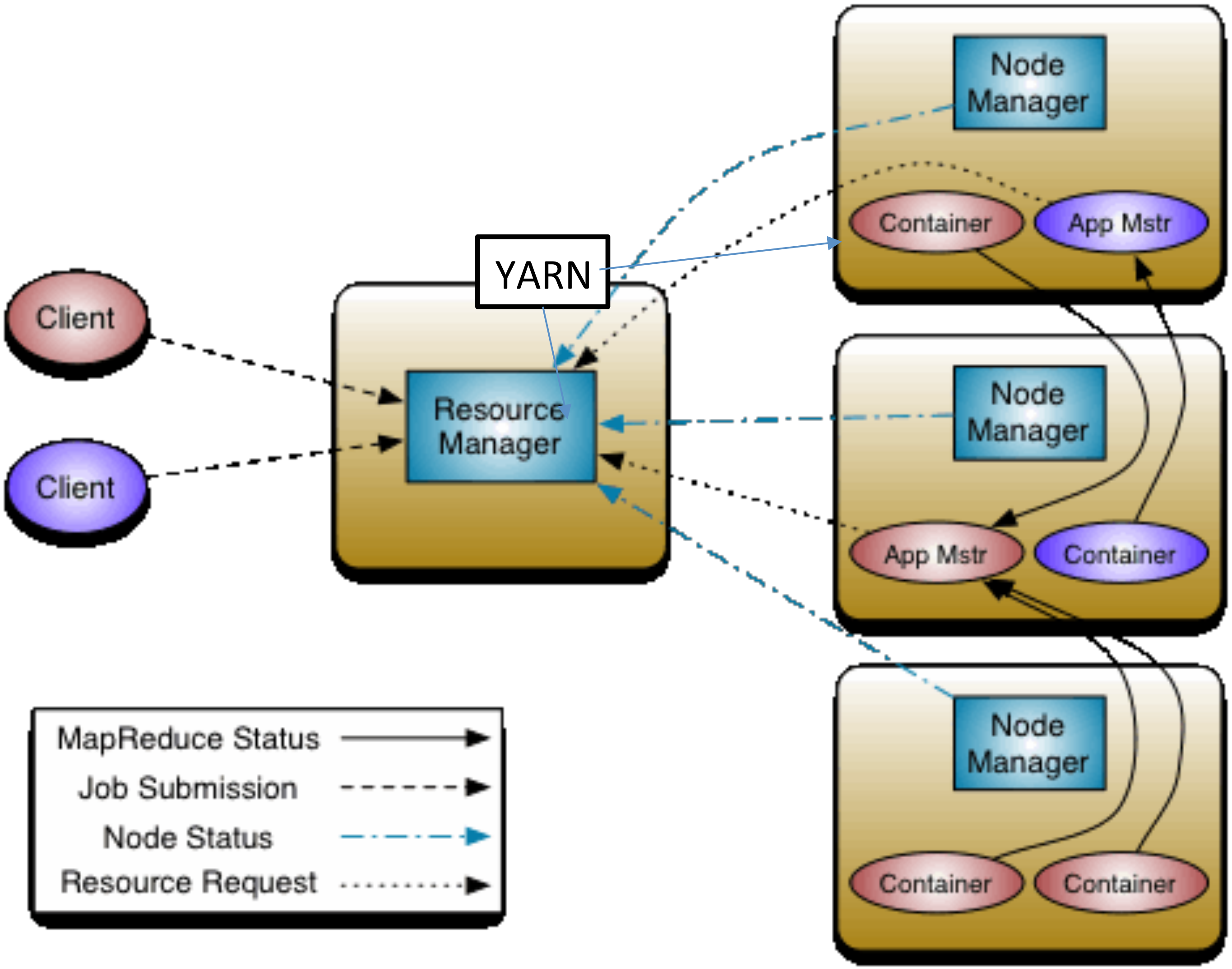
Storage Infrastructure

- **Problem:**
  - If nodes fail, how to store data persistently?
- **Answer:**
  - **Distributed File System:**
    - Provides global file namespace
    - Google GFS; Hadoop HDFS;
- **Typical usage pattern**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

What is Hadoop ?

- A scalable fault-tolerant distributed system for data storage and processing.
- Core Hadoop:
  - Hadoop Distributed File System (HDFS)
  - Hadoop YARN: Job Scheduling and Cluster Resource Management
  - Hadoop Map Reduce: Framework for distributed data processing.
- Open Source system with large community support.
  https://hadoop.apache.org/

Hadoop Architecture



Courtesy: http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html

HDFS

HDFS

- Assumptions
  - Hardware failure is the norm.
  - Streaming data access.
  - Write once, read many times.
  - High throughput, not low latency.
  - Large datasets.
- Characteristics:
  - Performs best with modest number of large files
  - Optimized for streaming reads
  - Layer on top of native file system.

HDFS

- Data is organized into file and directories.
- Files are divided into blocks and distributed to nodes.
- Block placement is known at the time of read
  - Computation moved to same node.
- Replication is used for:
  - Speed
  - Fault tolerance
  - Self healing.

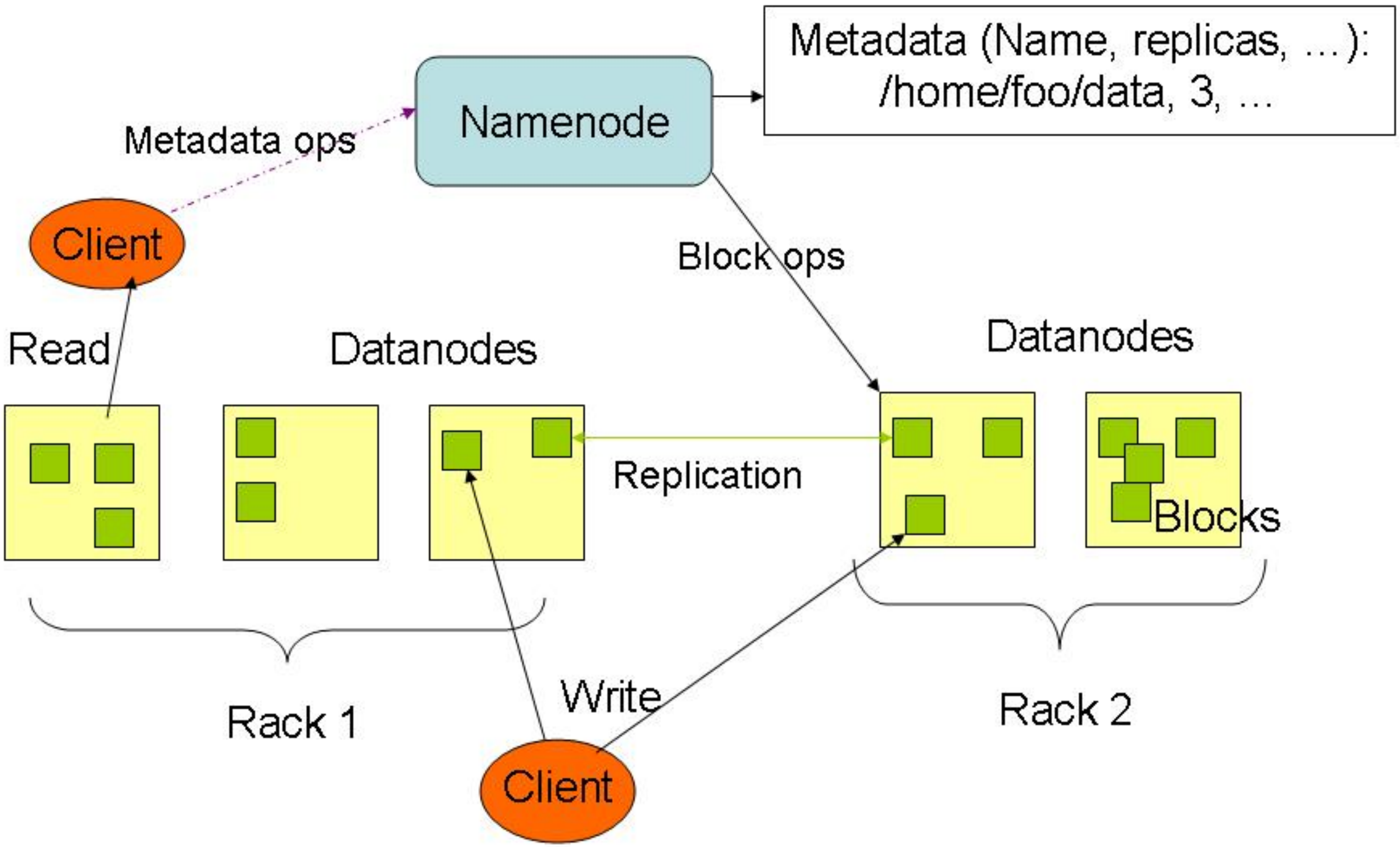
Goals of HDFS

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

Distributed File System

- **Single Namespace for entire cluster**
- **Data Coherency**
  – Write-once-read-many access model
  – Client can only append to existing files
- **Files are broken up into blocks**
  – Typically 128 MB block size
  – Each block replicated on multiple DataNodes
- **Intelligent Client**
  – Client can find location of blocks
  – Client accesses data directly from DataNode

HDFS Architecture



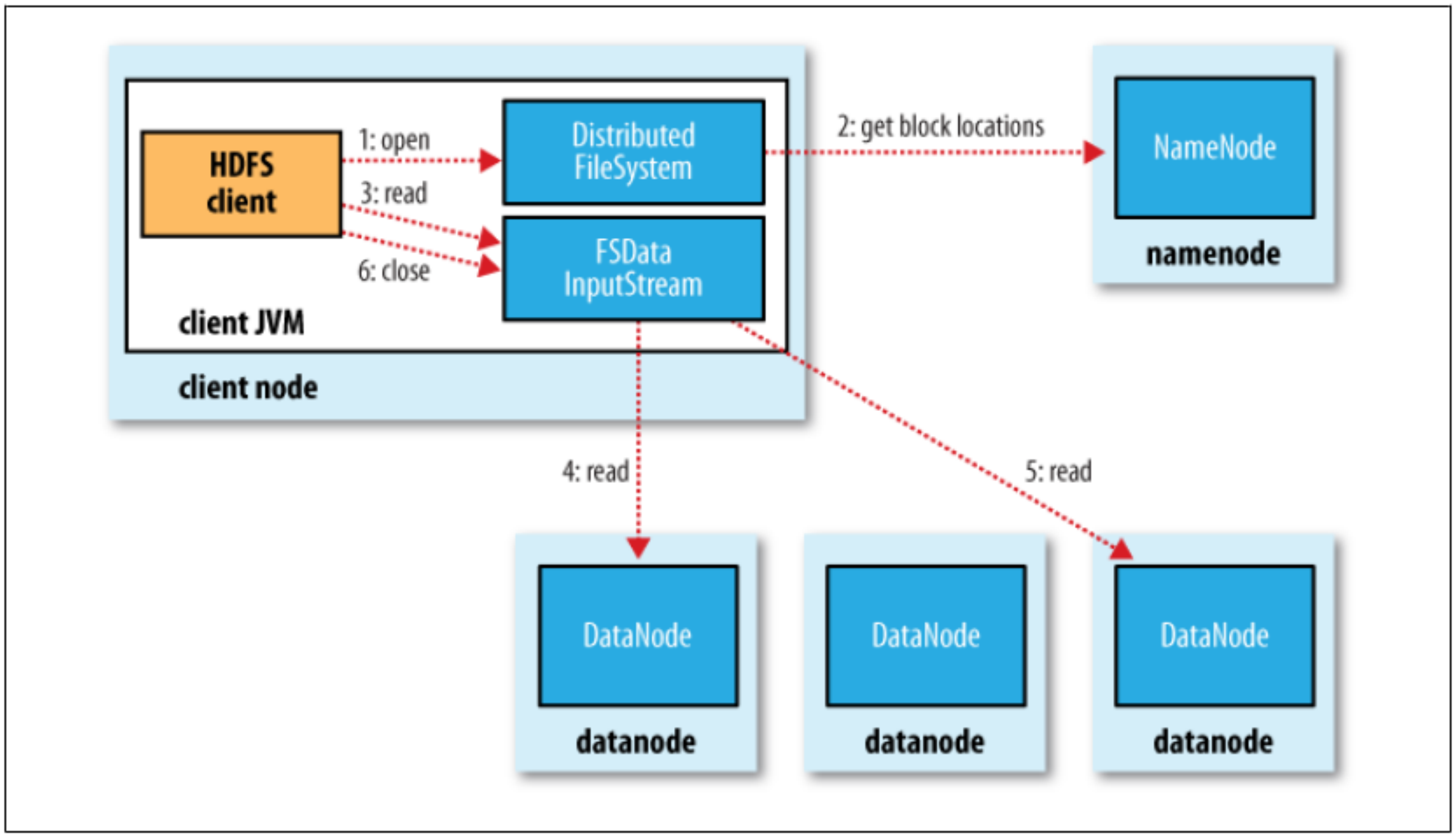
NameNode Metadata

- **Meta-data in Memory**
  - The entire metadata is in main memory
  - No demand paging of meta-data
- **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

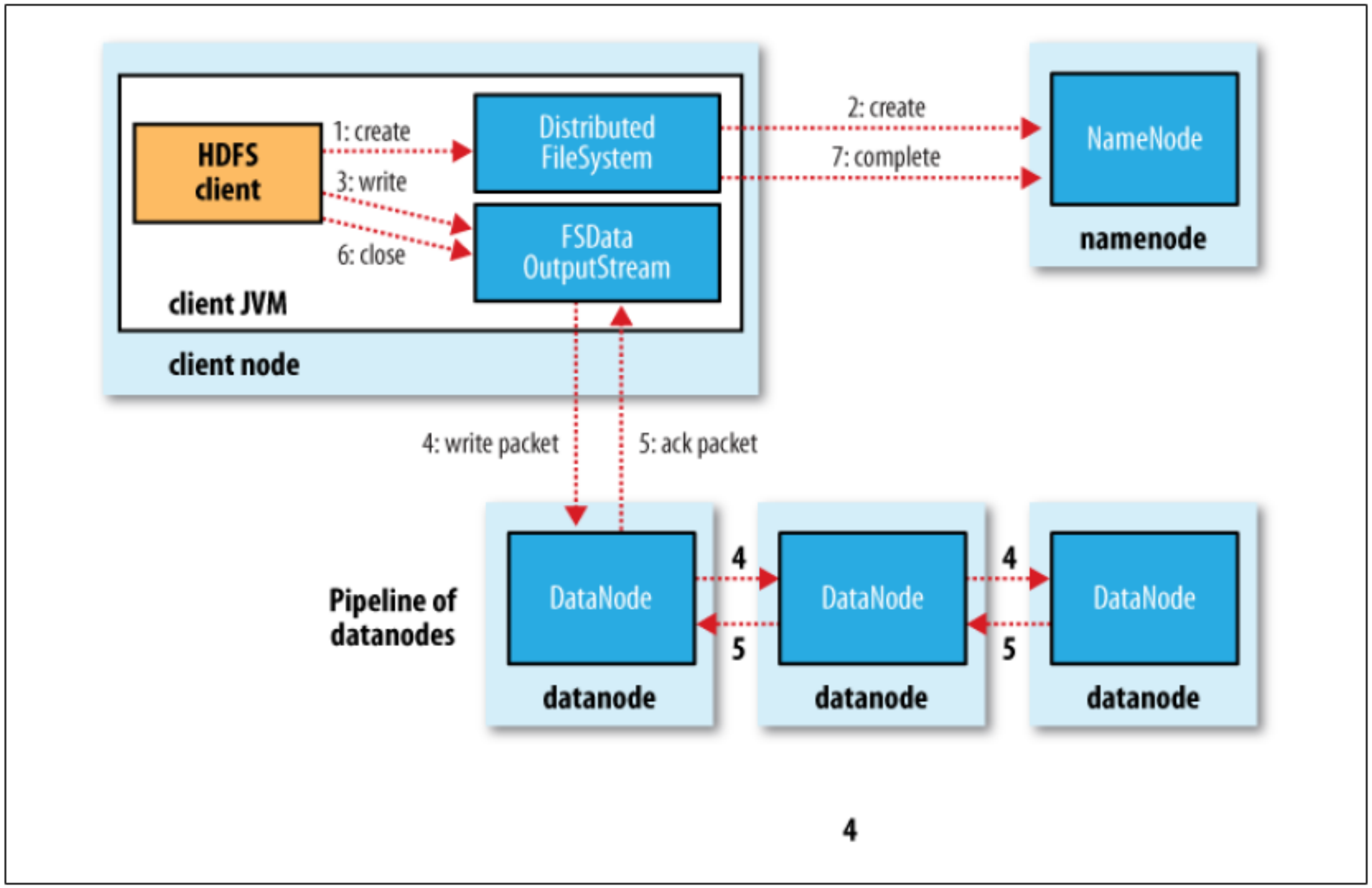
- **A Block Server**
  - Stores data in the local file system (e.g. ext3)
  - Stores meta-data of a block (e.g. CRC)
  - Serves data and meta-data to Clients
- **Block Report**
  - Periodically sends a report of all existing blocks to the NameNode
- **Facilitates Pipelining of Data**
  - Forwards data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

HDFS read client



Source: Hadoop: The Definitive Guide

HDFS write Client



Source: Hadoop: The Definitive Guide

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 replica**
- **Would like to make this policy pluggable**

NameNode Failure

- **A single point of failure**
- **Transaction Log stored in multiple directories**
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- **Need to develop a real HA solution**

Data Pipelining

- Client retrieves a list of DataNodes on which to place replicas of a block
- Client writes block to the first DataNode
- The first DataNode forwards the data to the next DataNode in the Pipeline
- When all replicas are written, the Client moves on to write the next block in file

MAP REDUCE

What is Map Reduce ?

- Method for distributing a task across multiple servers.
- Proposed by Dean and Ghemawat, 2004.
- Consists of two developer created phases:
  - Map
  - Reduce
- In between Map and Reduce is the Shuffle and Sort phase.
- User is responsible for casting the problem into map – reduce framework.
- Multiple map-reduce jobs can be “chained”.

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

Task: Word Count

Case 1:

– File too large for memory, but all <word, count> pairs fit in memory

Case 2:

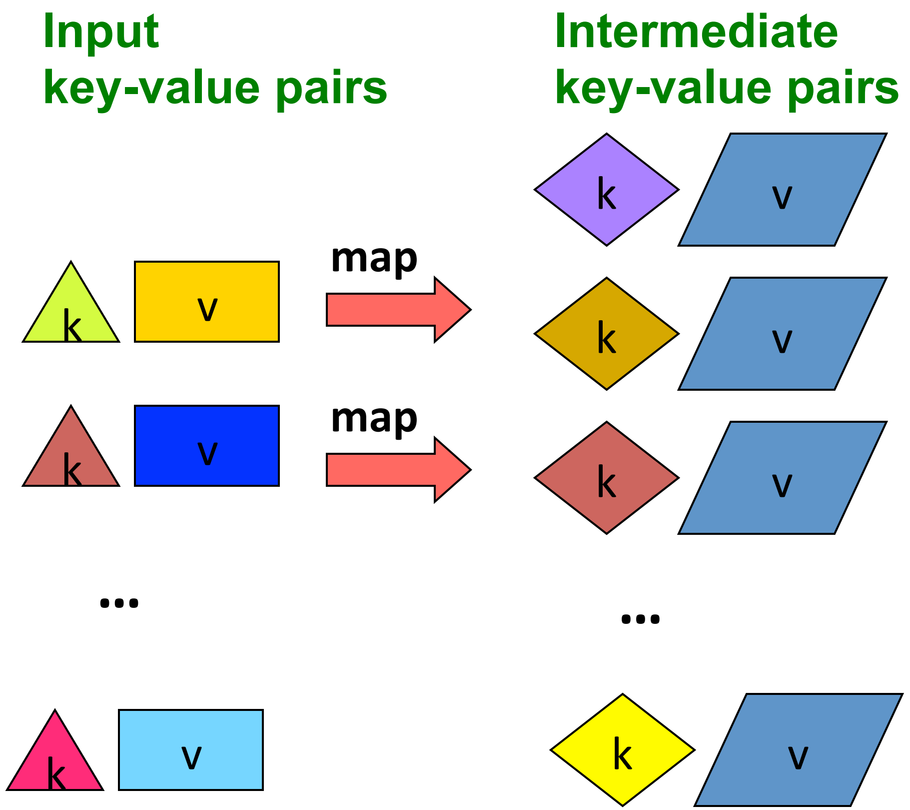
- Count occurrences of words:
  – `words(doc.txt) | sort | uniq -c`
    - where `words` takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of **MapReduce**
  – Great thing is that it is naturally parallelizable

MapReduce: Overview

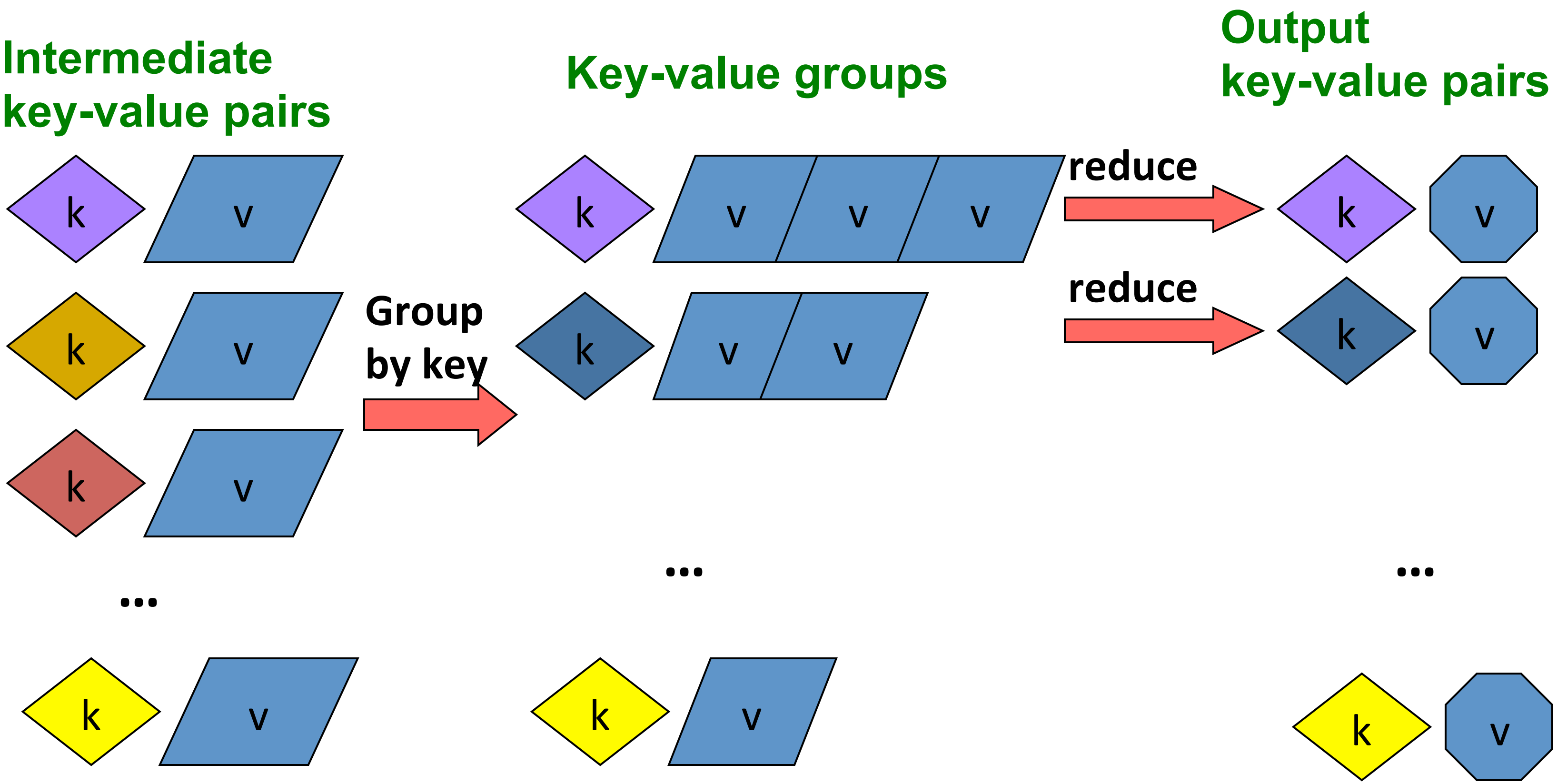
- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce: The Map Step



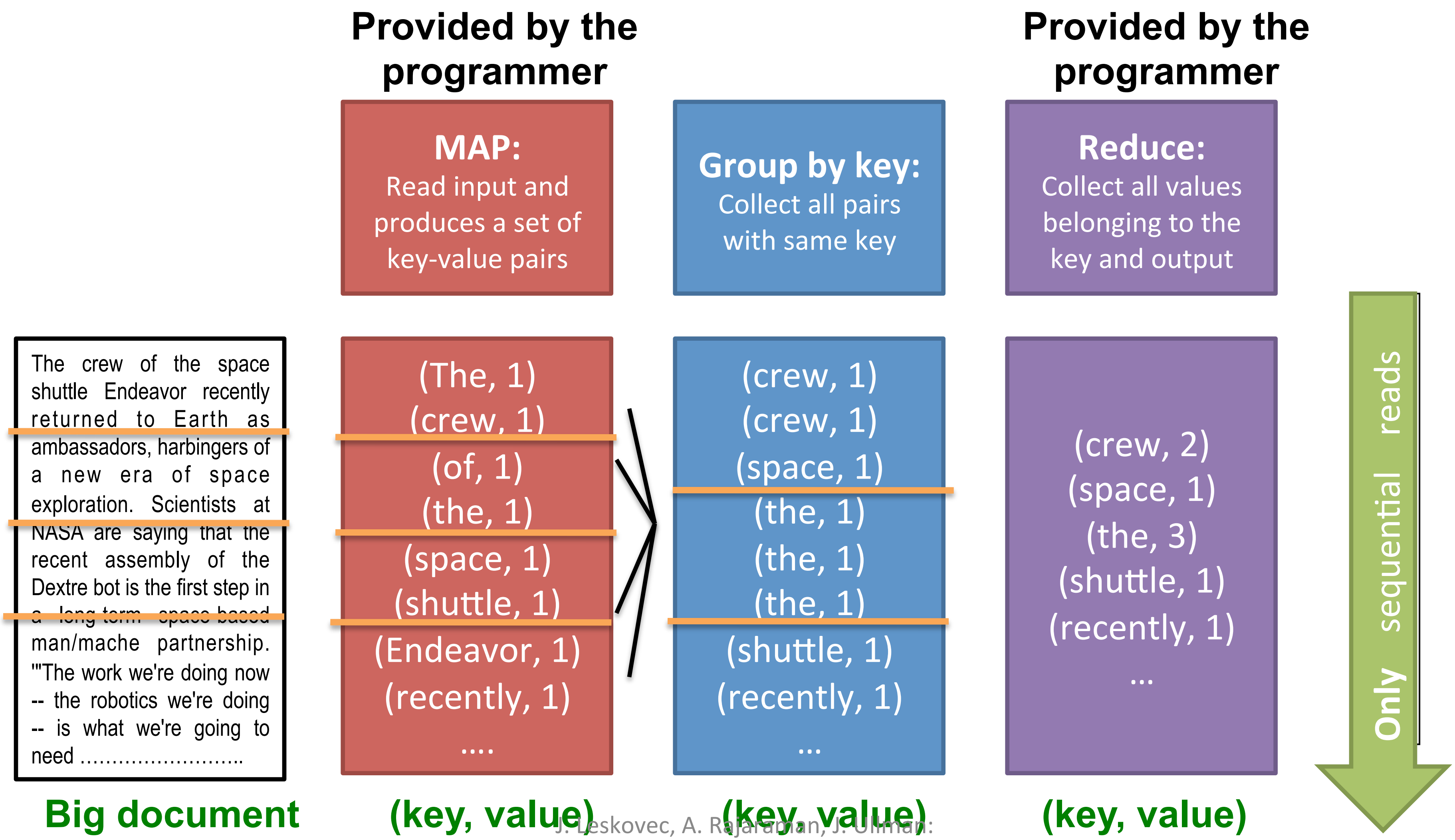
MapReduce: The Reduce Step



More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** → <k’, v’>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every *(k,v)* pair
  - **Reduce(k’, <v’>*)** → <k’, v’’>*
    - **All values *v’* with same key *k’* are reduced together and processed in *v’* order**
    - There is one Reduce function call per unique key *k’*

MapReduce: Word Counting



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 Phase

- User writes the mapper method.
- Input is an unstructured record:
  - E.g. A row of RDBMS table,
  - A line of a text file, etc
- Output is a set of records of the form: <key, value>
  - Both key and value can be anything, e.g. text, number, etc.
  - E.g. for row of RDBMS table: <column id, value>
  - Line of text file: <word, count>

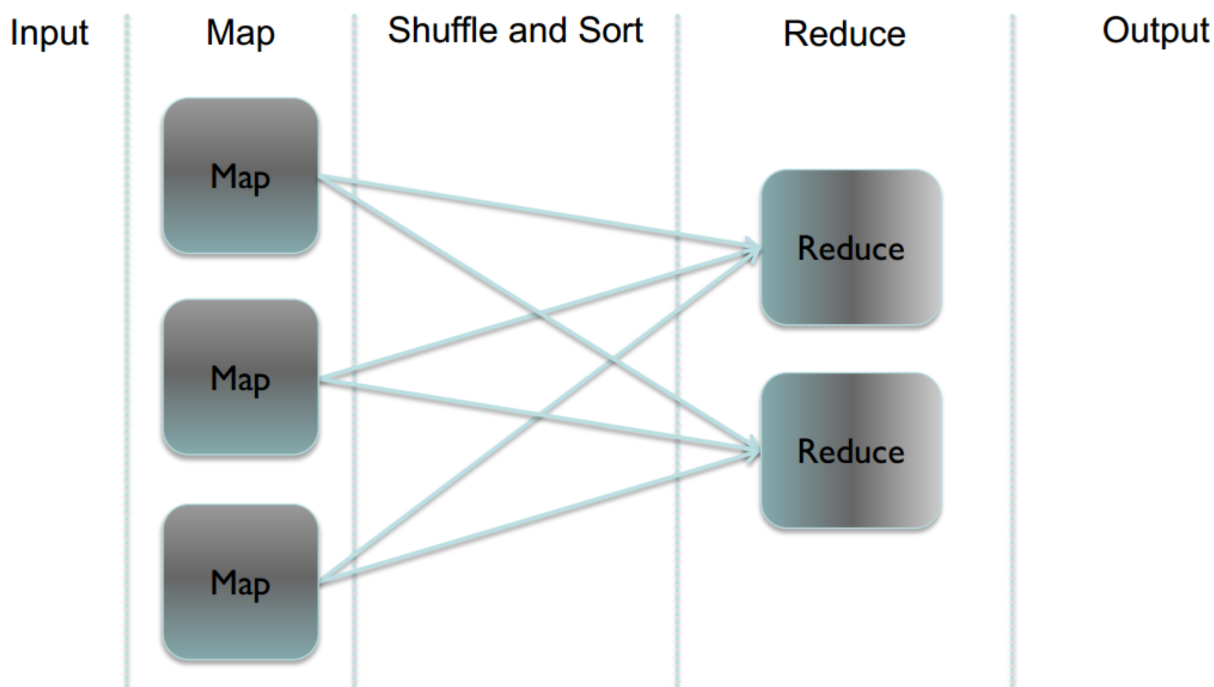
Shuffle/Sort phase

- Shuffle phase ensures that all the mapper output records with the same key value, goes to the same reducer.
- Sort ensures that among the records received at each reducer, records with same key arrives together.

Reduce phase

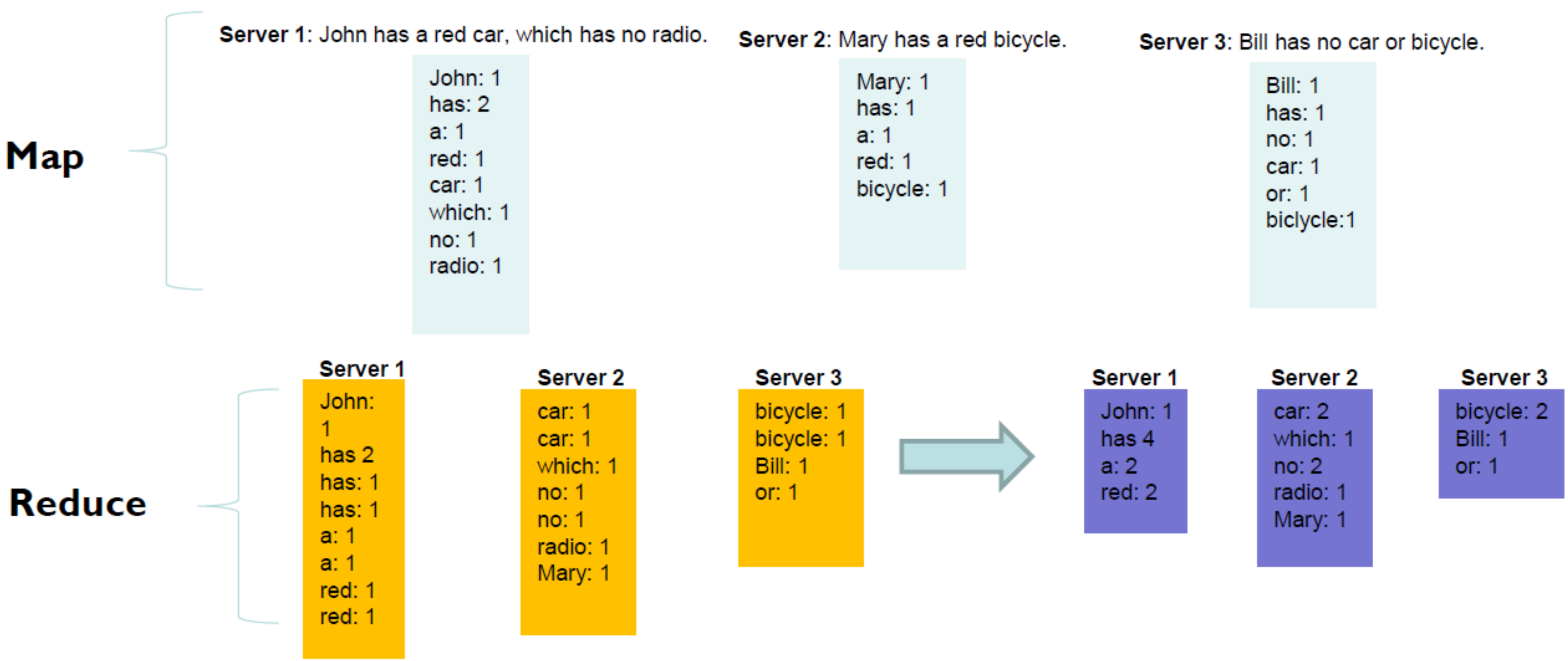
- Reducer is a user defined function which processes mapper output records with some of the keys output by mapper.
- Input is of the form <key, value>
  - All records having same key arrive together.
- Output is a set of records of the form <key, value>
  - Key is not important

Parallel picture

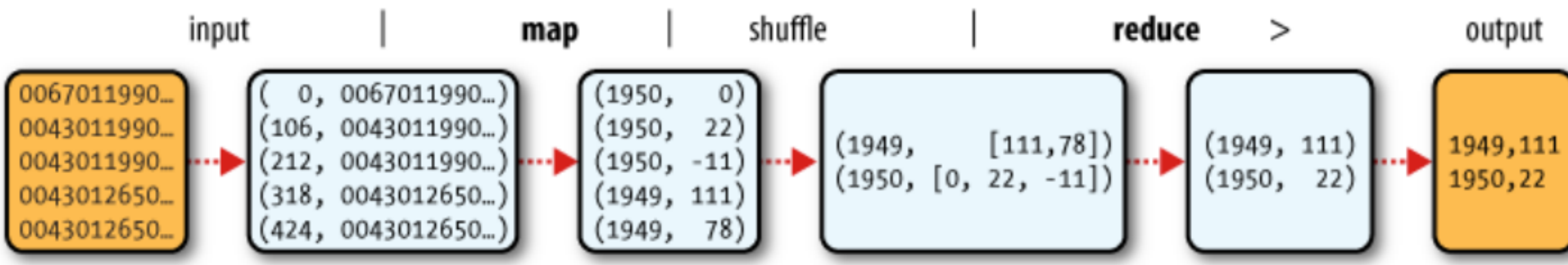


Example

- Word Count: Count the total no. of



Map Reduce

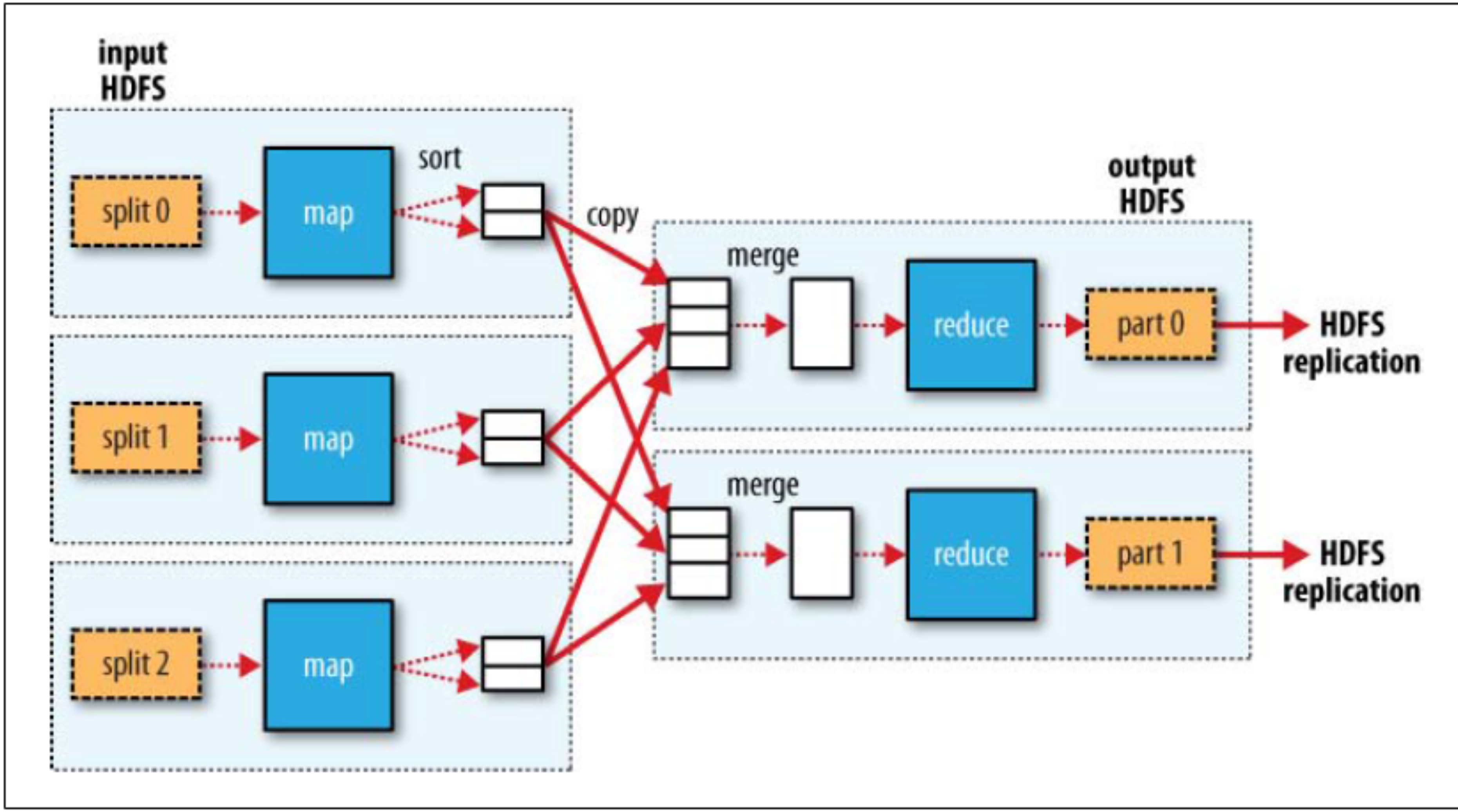


What was the max/min temperature for the last century ?

Hadoop Map Reduce

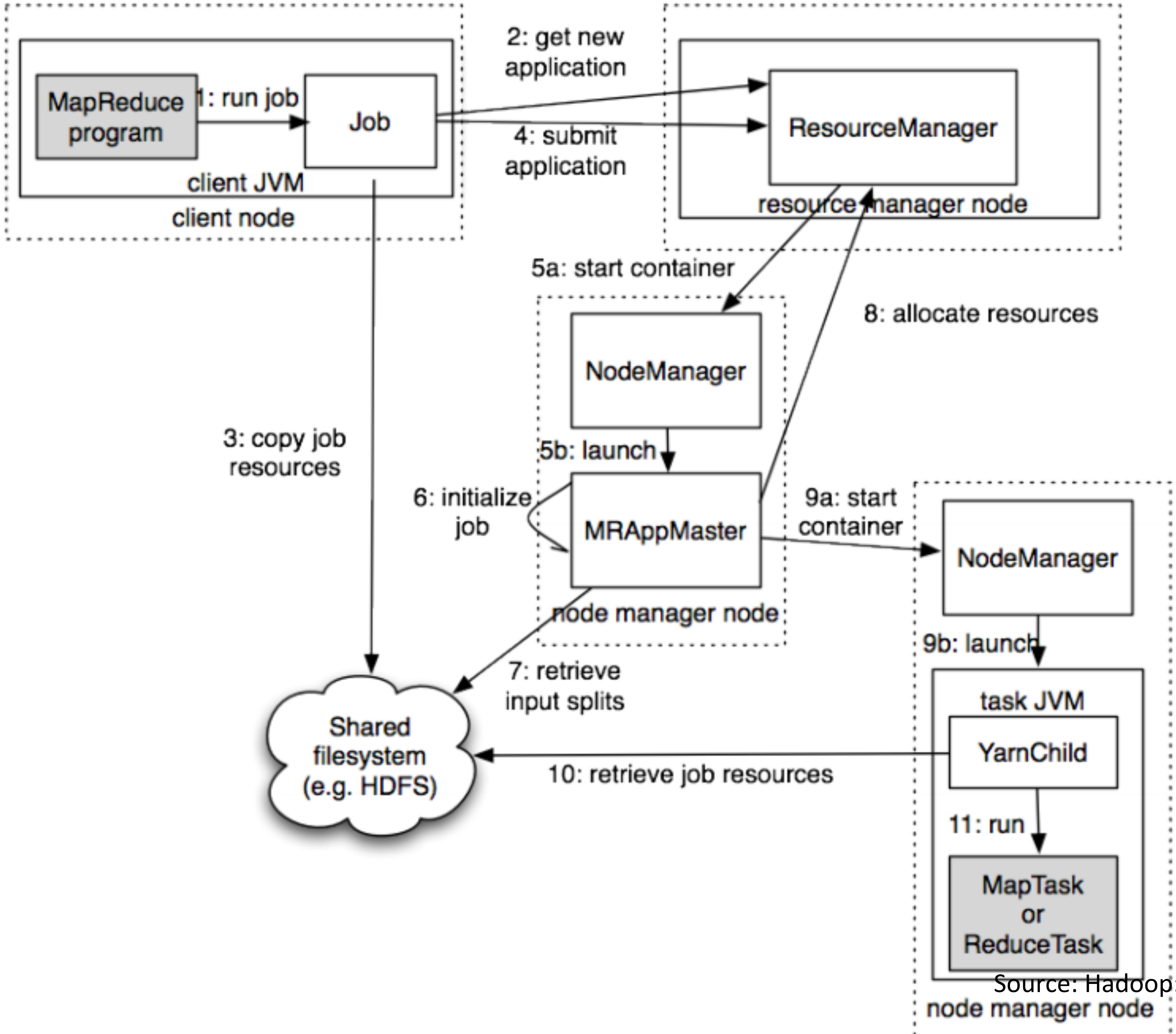
- Provides:
  - Automatic parallelization and Distribution
  - Fault Tolerance
  - Methods for interfacing with HDFS for colocation of computation and storage of output.
  - Status and Monitoring tools
  - API in Java
  - Ability to define the mapper and reducer in many languages through Hadoop streaming.

Hadoop MR Data Flow



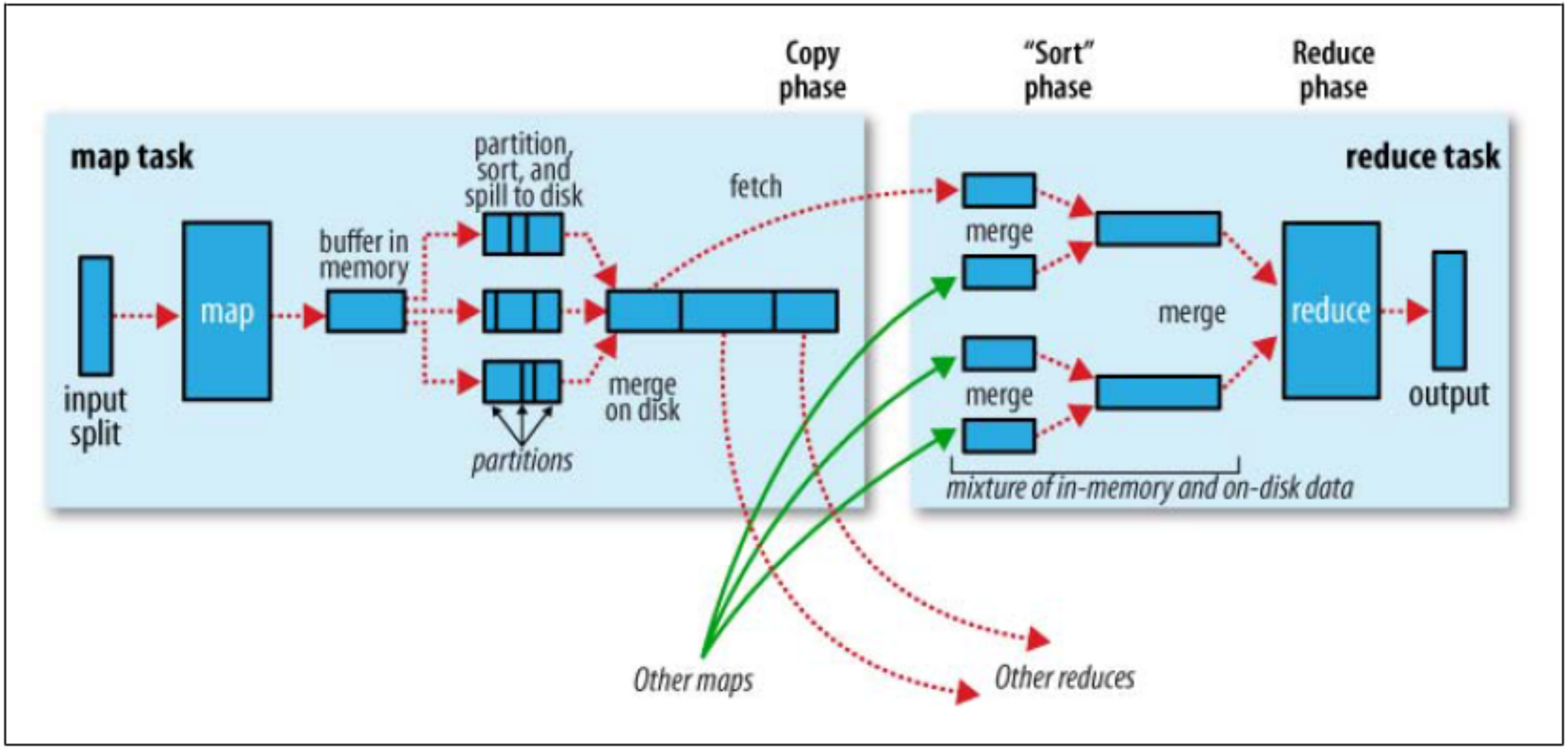
Source: Hadoop: The Definitive Guide

Hadoop(v2) MR job



Source: Hadoop: The Definitive Guide

Shuffle and sort



Source: Hadoop: The Definitive Guide

Data Flow

- **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

Fault tolerance

- Comes from scalability and cost effectiveness
- HDFS:
  - Replication
- Map Reduce
  - Restarting failed tasks: map and reduce
  - Writing map output to FS
  - Minimizes re-computation

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

Failures

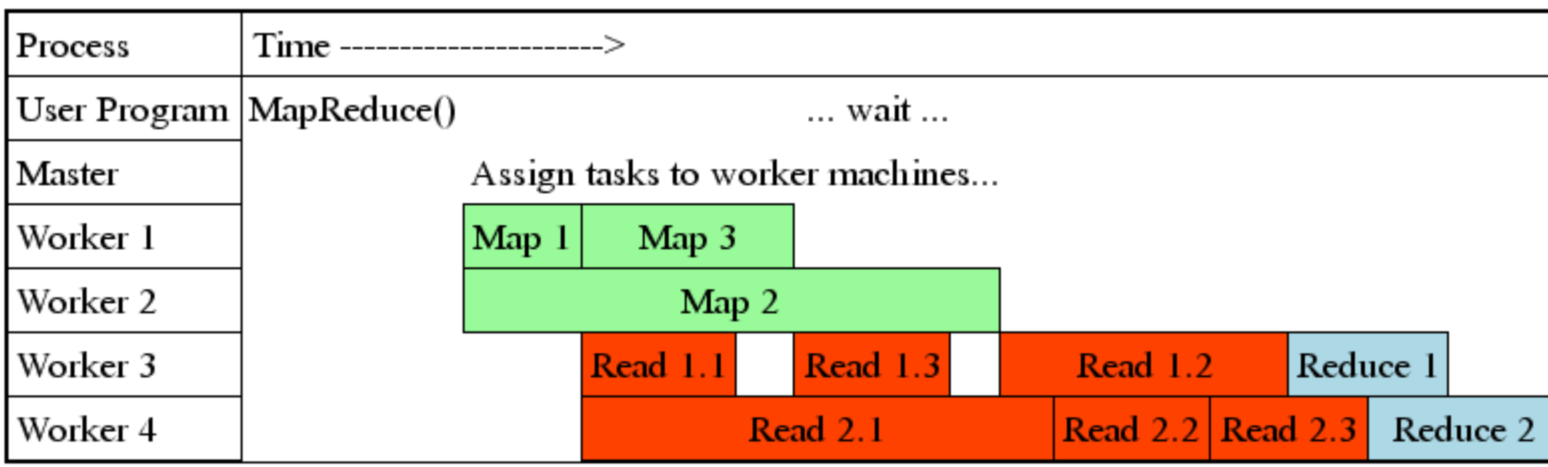
- Task failure
  - Task has failed – report error to nodemanager, appmaster, client.
  - Task not responsive, JVM failure – Nodemanager restarts tasks.
- Application Master failure
  - Application master sends heartbeats to resourcemanager.
  - If not received, the resource manager retrives job history of the run tasks.
- Node manager failure

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

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

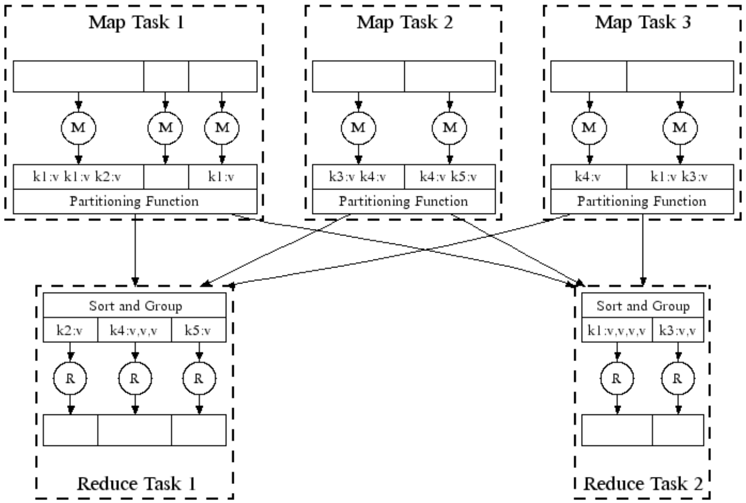


Refinements: Backup Tasks

- **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

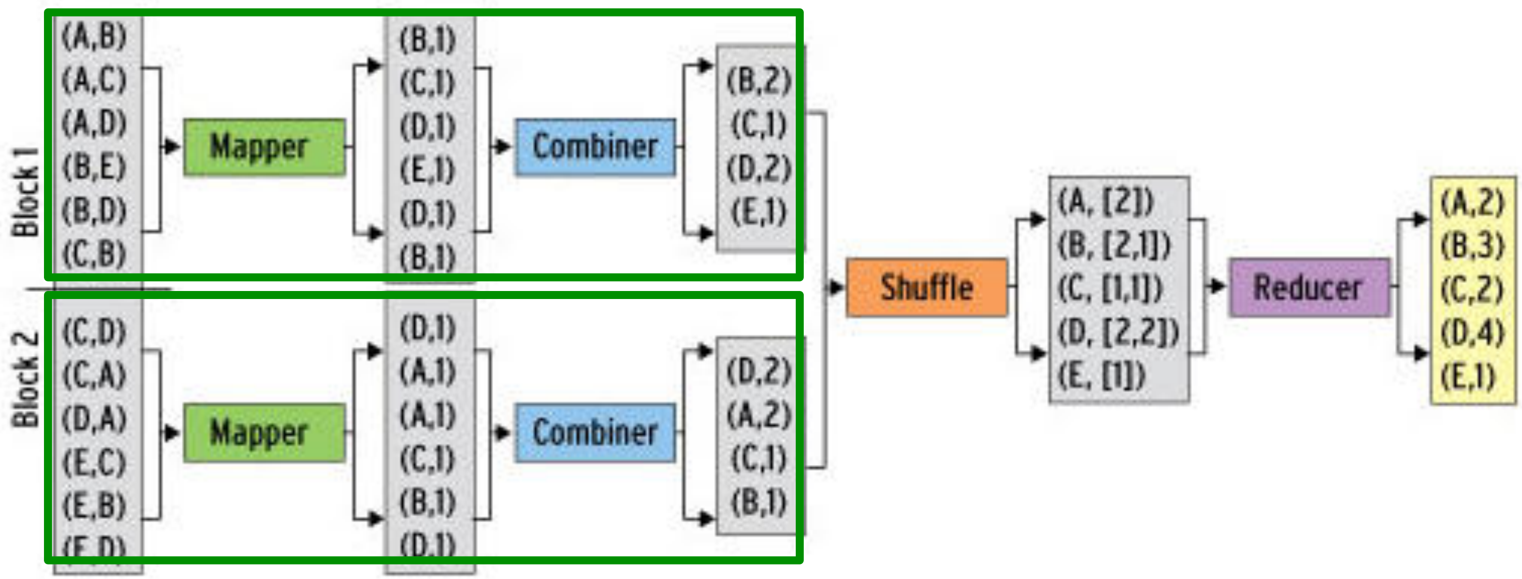
Refinement: Combiners

- Often a Map task will produce many pairs of the form *(k,v_1), (k,v_2), …* for the same key *k*
  - E.g., popular words in the word count example
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function
- Works only if reduce function is commutative and associative



Refinement: Combiners

- **Back to our word counting example:**
  - Combiner combines the values of all keys of a single mapper (single machine):



  - Much less data needs to be copied and shuffled!

Refinement: Partition Function

- **Want to control how keys get partitioned**
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- **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

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

$\bowtie$

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Map-Reduce Join

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*

- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.