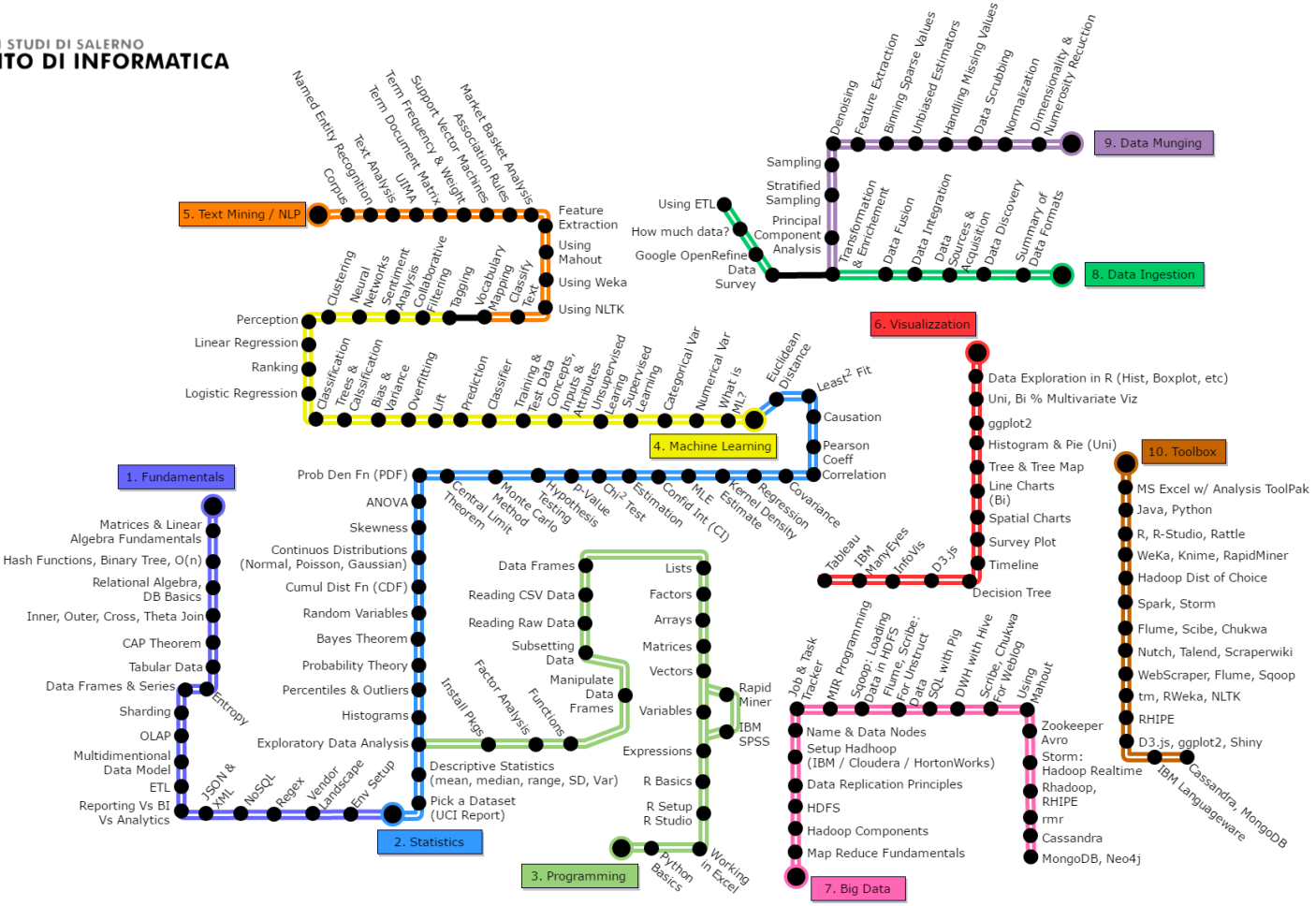
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Fondamenti di Data Science & Machine Learning

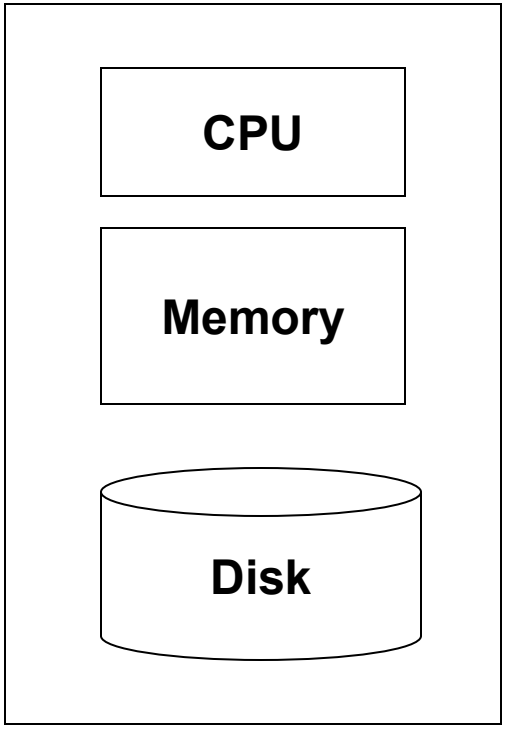
Map Reduce

Prof. Giuseppe Polese, aa 2024-25

MapReduce

- **Large scale computing** for **data mining**
- **Challenges:**
  - How to distribute computation?
  - Distributed/parallel programming is hard
- **Map-reduce** addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data

Single Node Architecture



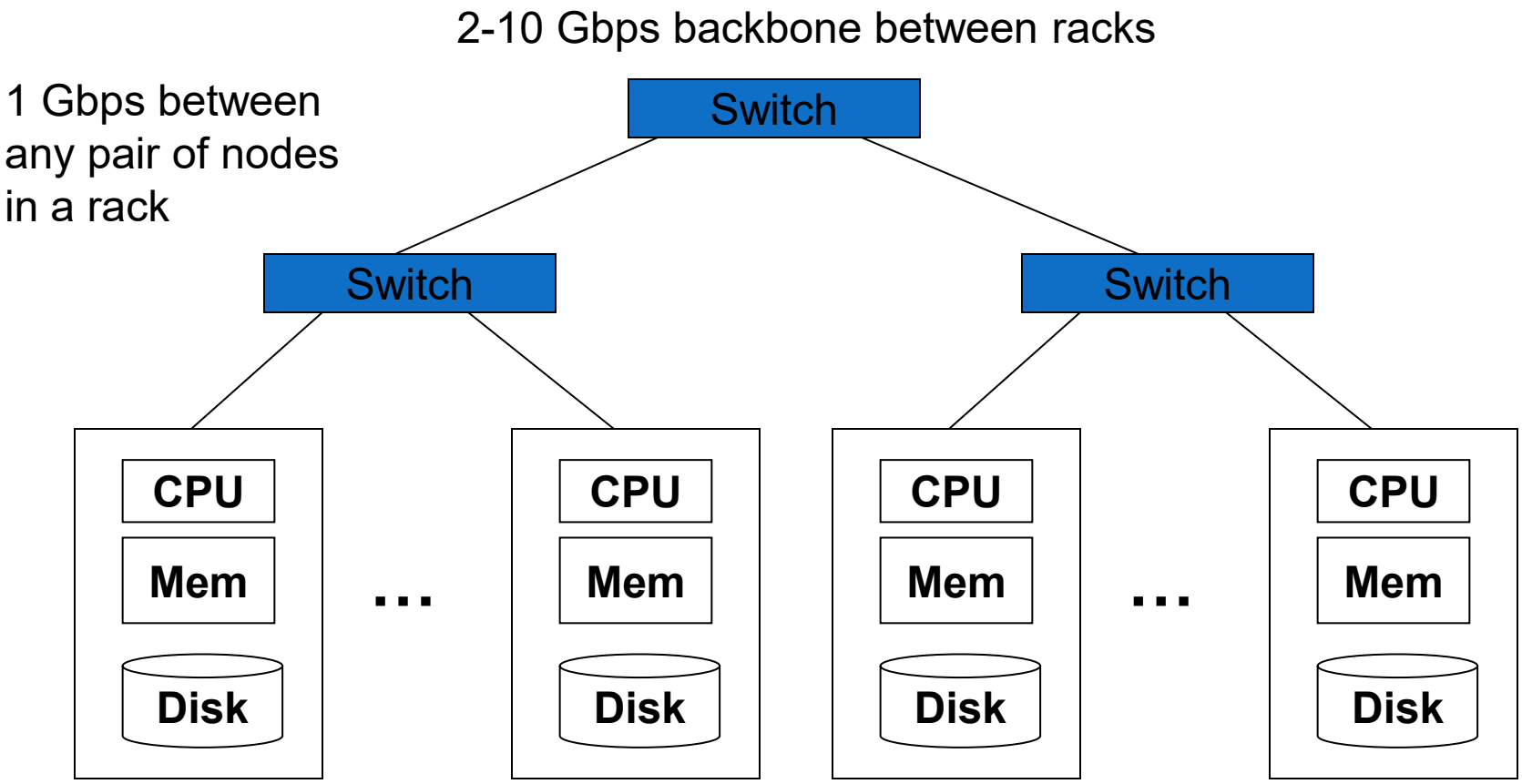
Machine Learning, Statistics

“Classical” Data Mining

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



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!

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
  - Google's computational/data manipulation model
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS. Kosmix: Cloudstore
  - **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

Usage Pattern

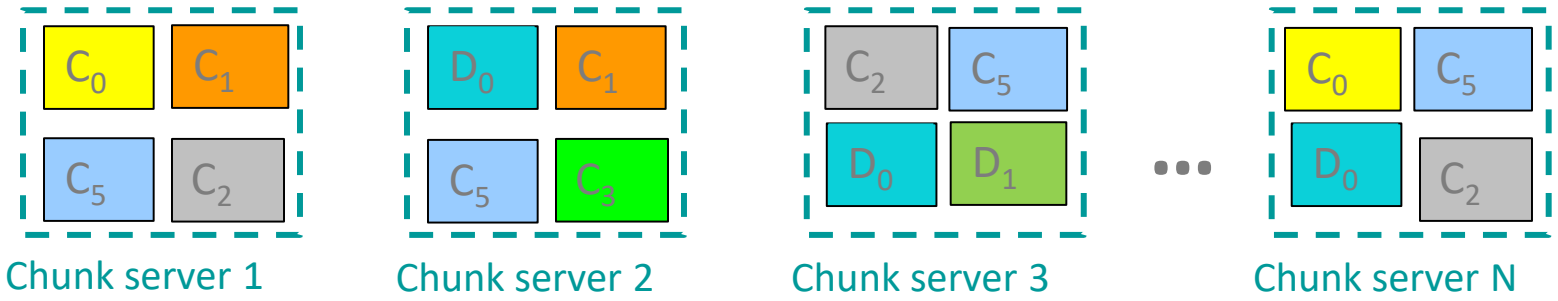
- **Huge files (100s of GB to TB)**
  - There is no point using a DFS for small files
- **Data is rarely updated**
  - They are read for some calculation, and possibly new data is appended to files from time to time.
  - For example, an airline reservation system would not be suitable for a DFS, even if the data were very large, because the data is changed frequently.

Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks of 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- **Master node**
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might be replicated
- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

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

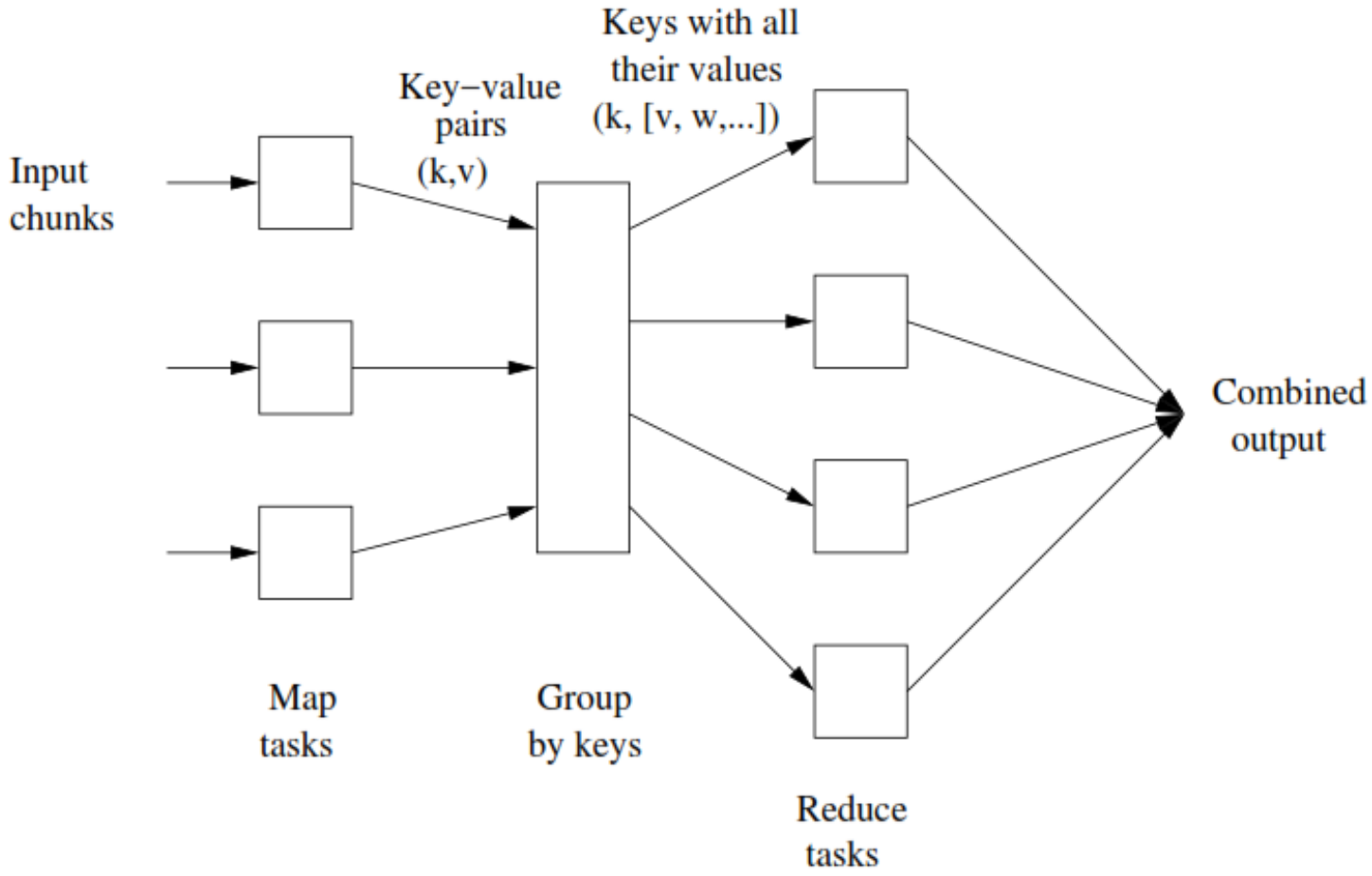
Programming Model: MapReduce

- MapReduce is a style of computing implemented in several systems, including Google's and the open-source implementation Hadoop from the Apache Foundation, along with the HDFS file system.
- A MapReduce implementation allows to manage many large-scale computations in a way that is tolerant of hardware faults.
- All you need to write are 2 functions, Map and Reduce, while the system manages the parallel execution, coordination of tasks that execute Map or Reduce, dealing with the possibility that some tasks fail.

MapReduce Computation

1. Several Map tasks each are given one or more chunks from a distributed file system, turning them into a sequence of key-value pairs according to code written by the user for the Map function.
2. The key-value pairs from each Map task are collected by a master controller, sorted by key, and divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.
3. The Reduce tasks combine all the values associated with a key according to the code written by the user for the Reduce function.

MapReduce Computation Schema



Sample Applications of 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:

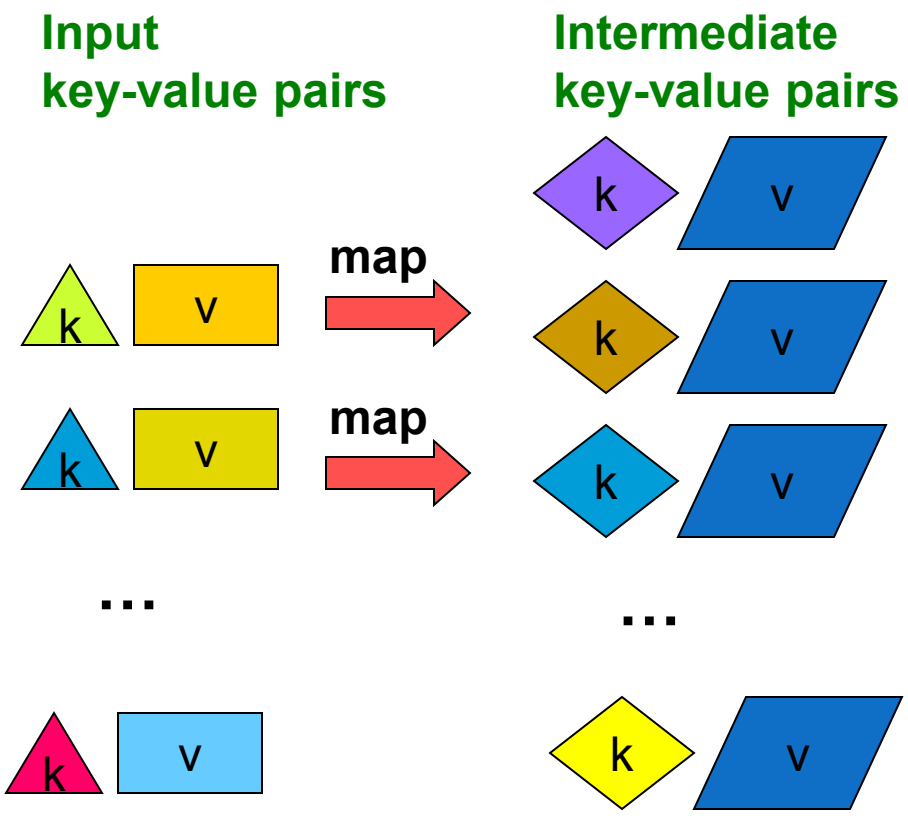
- Count occurrences of words:
  - `words(doc.txt) | sort | uniq -c`
    - where **`words`** takes a file and outputs the words in it, one per line
- Case 2 captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable

MapReduce: Overview

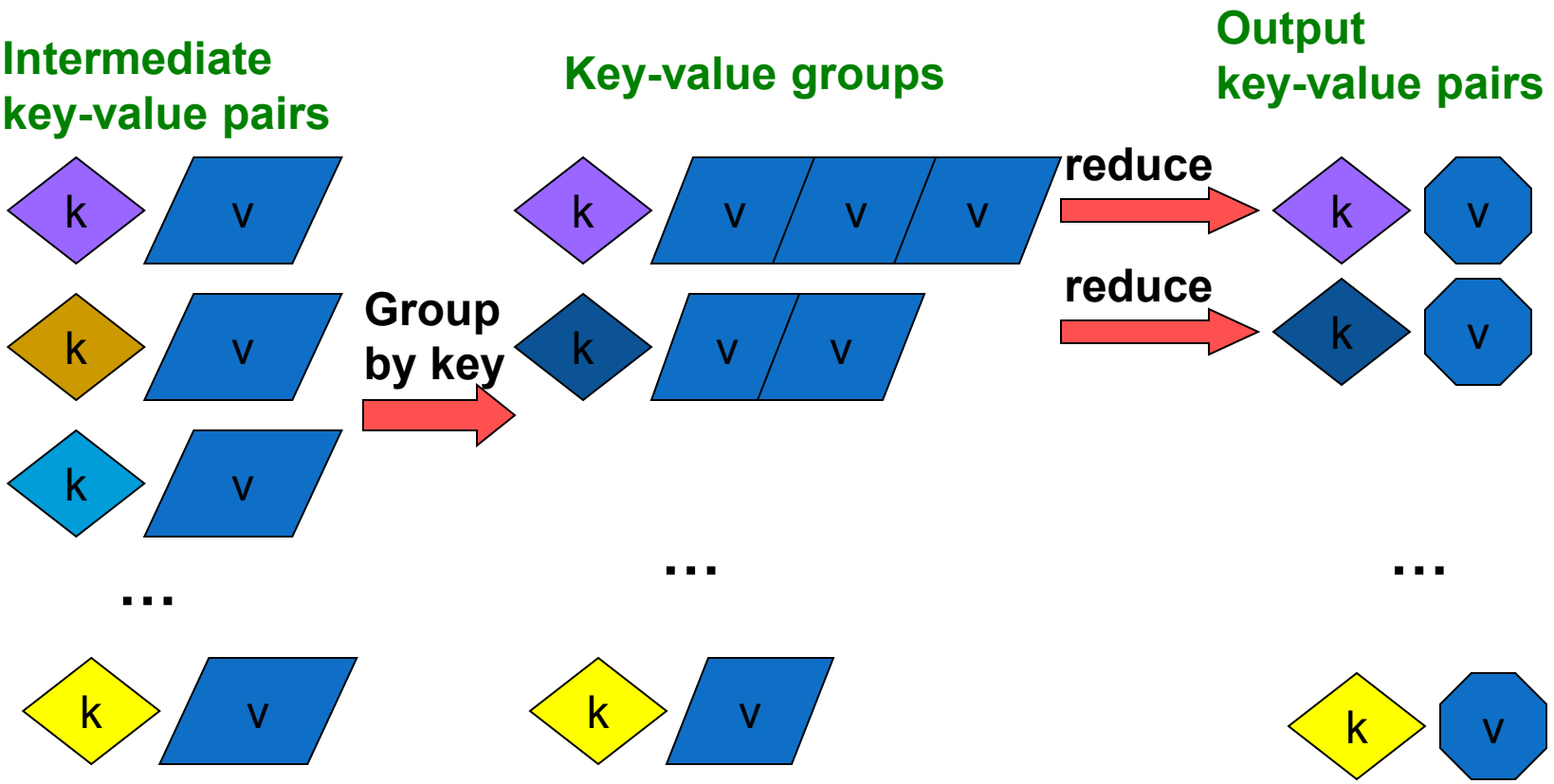
- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

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

MapReduce: The Map Step



MapReduce: The Reduce Step



MapReduce: Map Tasks

- Input files for a Map task are viewed as consisting of elements: a tuple or a document.
- A chunk is a collection of elements, and no element is stored across two chunks.
- The Map function takes an input element and produces zero or more key-value pairs.
- Keys do not have to be unique. A Map task can produce several key-value pairs with the same key.

Grouping by Keys

- As the Map tasks are completed, the system groups key-value pairs by key, regardless of what the Map and Reduce tasks do, and the values associated with each key are formed into a list of values.
- The master controller process knows how many Reduce tasks the user has specified (say r).
- The master controller applies a hash function to keys and produces bucket numbers from 0 to r – 1, and each key-value pair is put in one of r local files, each destined to one of the Reduce tasks.

Assignments to Reduce Tasks

- For each key k, the master controller sends the input to the Reduce task that handles k as a pair of the form $(k, [v_1, v_2, \ldots, v_n])$.
- $(k, v_1), (k, v_2), \ldots, (k, v_n)$ are all the key-value pairs with key k coming from all the Map tasks.

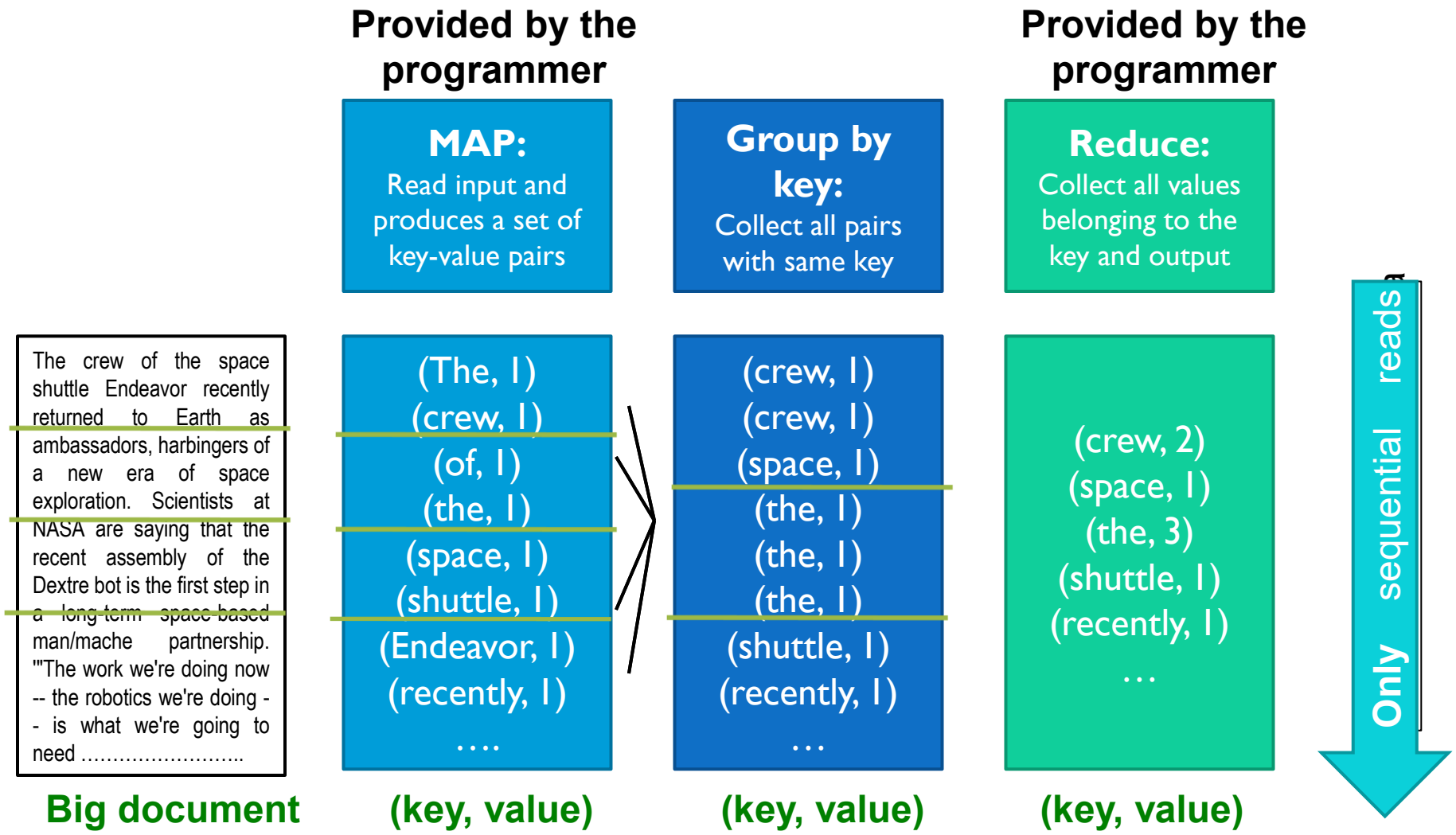
MapReduce: Reduce Tasks

- The Reduce function processes the input pair of a key and its list of values and outputs a sequence of zero or more key-value pairs.
- These key-value pairs can be of a type different from those sent from Map tasks, but often are the same type.
- The application of the Reduce function to a key and its associated list of values is named reducer.
- A Reduce task executes one or more reducers. Outputs from all the Reduce tasks are merged into a single file.

More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** $\rightarrow$ <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'>*)** $\rightarrow$ <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-Reduce: Execution

- The user program forks a Master controller process and some Worker processes at different compute nodes.
- A Worker handles either Map tasks (Map worker) or Reduce tasks (Reduce worker), but not both.
- The Master creates some Map tasks and some Reduce tasks (as many as specified by the user program). The Master assigns these tasks to Worker processes.
- It is reasonable to create one Map task for every chunk of the input file(s), but fewer Reduce tasks, since each Map task must create an intermediate file for each Reduce task, hence the number of files might explode.

Map-Reduce: Execution (2)

- The Master monitors the status of each Map and Reduce task (idle, executing at a Worker, or completed).
- A Worker process reports to the Master when it finishes a task, and it gets a new scheduled task from it.
- Each Map task is assigned one or more chunks of the input file(s) and executes the user written code on it.
- The Map task creates a file for each Reduce task on the local disk of the Worker executing the Map task.
- The Master is informed of location and sizes of these files, and the Reduce task to which each is destined.

Map-Reduce: Execution (3)

- When the Master assigns a Reduce task to a Worker process, the task is given all the files that form its input.
- The Reduce task executes the user written code and writes its output to a file that is part of the surrounding distributed file system.
- The Master periodically pings Worker processes to detect possible failures at compute nodes.

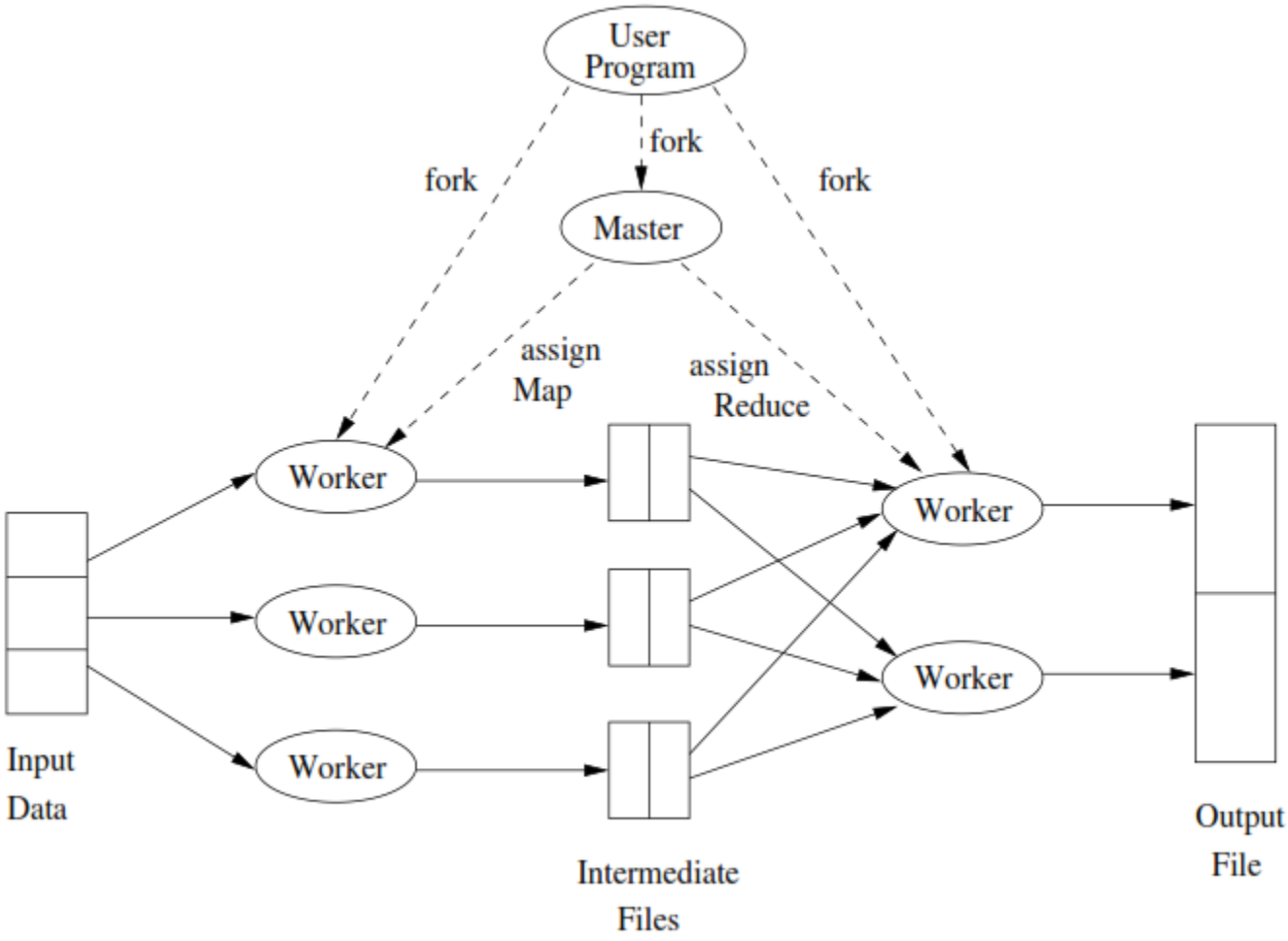
Coping with Node Failures

- If the Master detects a failure at a compute node at which a Map worker resides, all the Map tasks assigned to the Worker must be redone, even if they had completed.
- This because the output destined to Reduce tasks is on the failed node, hence it is not available to them.
- The Master sets the status of these Map tasks to idle and will schedule them on a different Worker when one becomes available.

Coping with Node Failures (2)

- The Master must also inform each Reduce task that the location of its input coming from the redone Map task has changed.
- Dealing with a failure at the node of a Reduce worker is simpler.
- The Master simply sets the status of its currently executing Reduce tasks to idle.
- These will be rescheduled on another reduce worker later.

Execution of Map-Reduce Program



Map-Reduce: Environment

Map-Reduce environment takes care of:

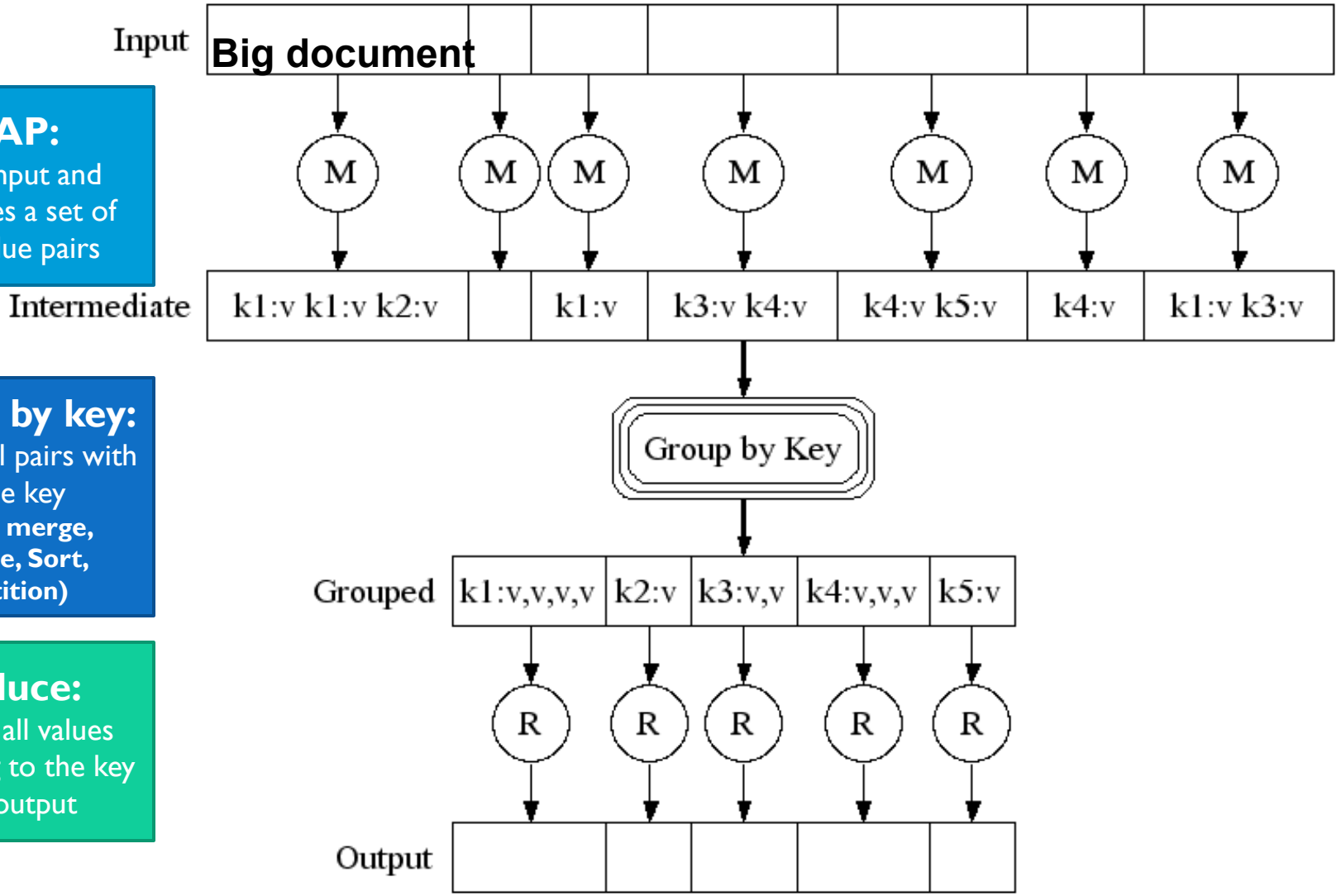
- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine failures
- Managing required inter-machine communication

Map-Reduce: A diagram

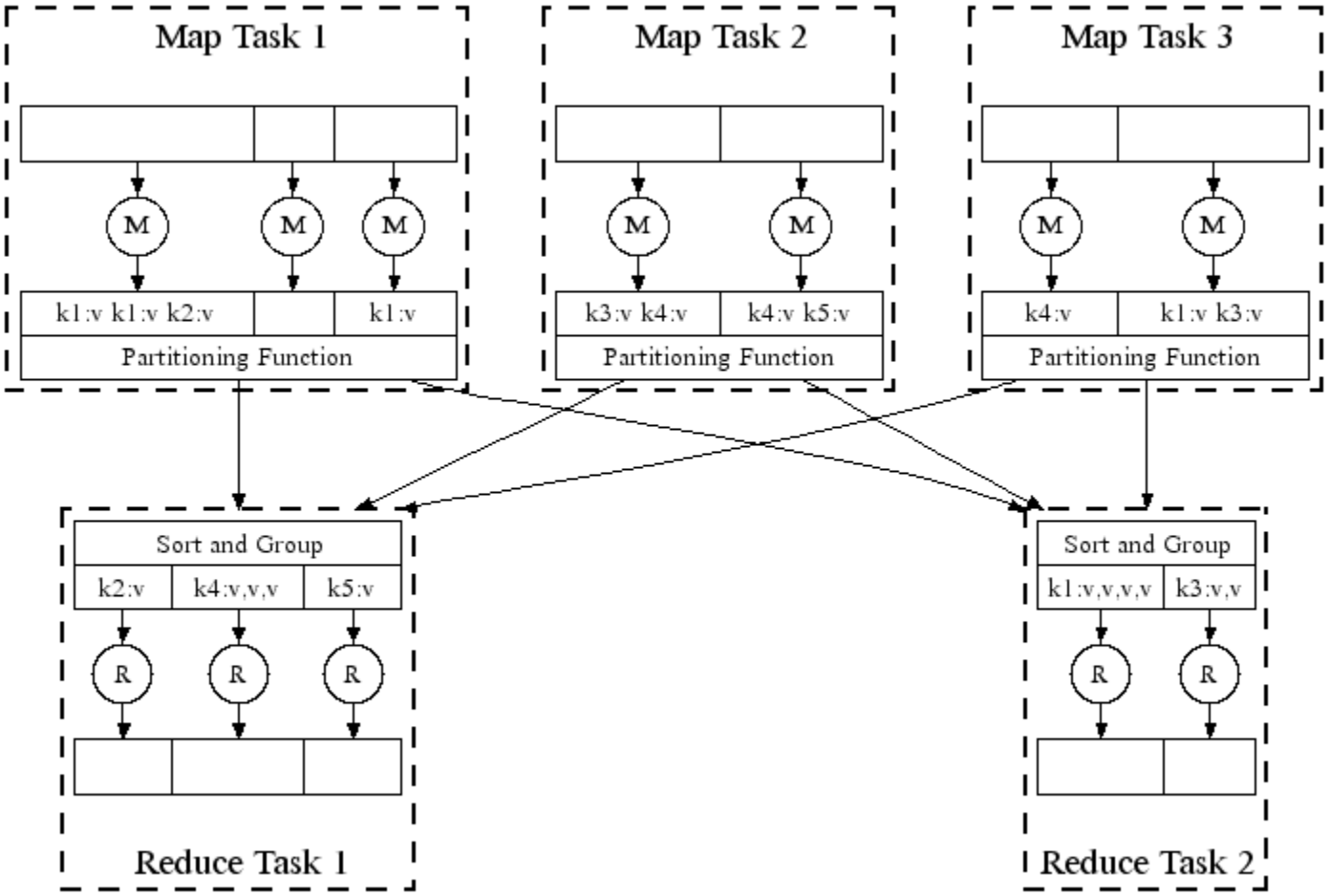
MAP:
Read input and produces a set of key-value pairs

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

Reduce:
Collect all values belonging to the key and output



Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

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

Increasing Reduce Task Parallelism

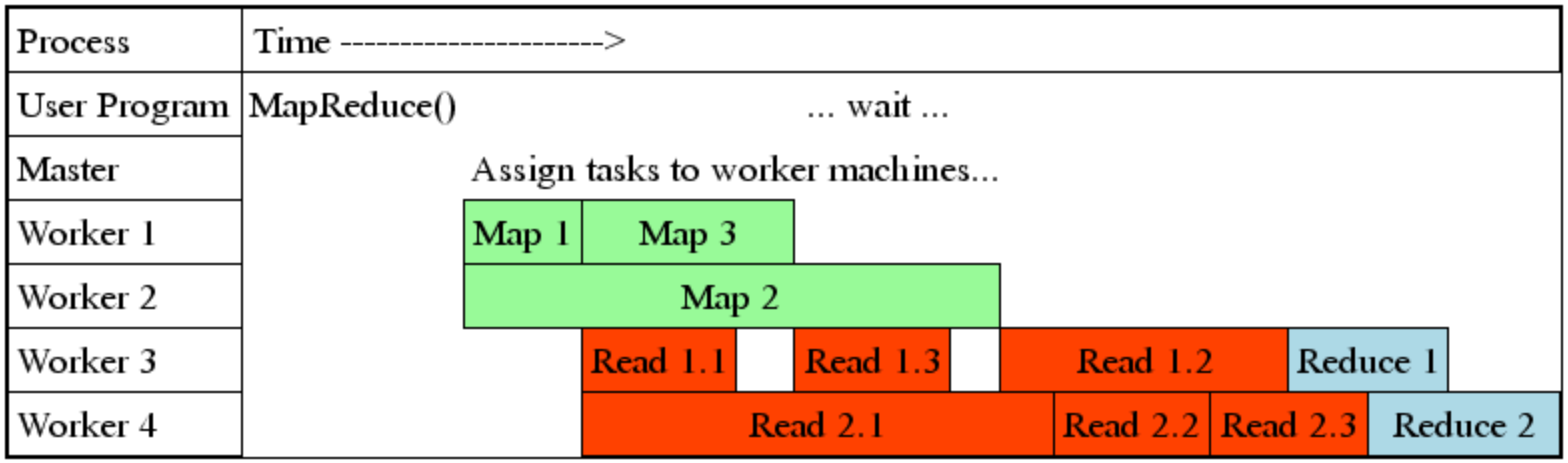
- Having one Reduce task to execute each reducer (i.e., a single key and its associated value list) would maximize parallelism, since each task could execute at a different compute node.
- This is the best! There is overhead associated with each task we create, so better keep the number of Reduce tasks lower than the number of different keys.
- There are far more keys than available compute nodes, so there is no benefit from a huge number of Reduce tasks.

Reduce Tasks Skew

- Since the value lists of different keys can vary in length, making each reducer a separate task would yield skew – a significant difference in the amount of execution time.
- We can reduce skew by using fewer Reduce tasks than the number of reducers.
- If keys are sent randomly to Reduce tasks, we can expect some averaging of the total execution time of Reduce tasks.
- We can further reduce skew by using more Reduce tasks than compute nodes (i.e., one executing a long task and one executing many small tasks sequentially).

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

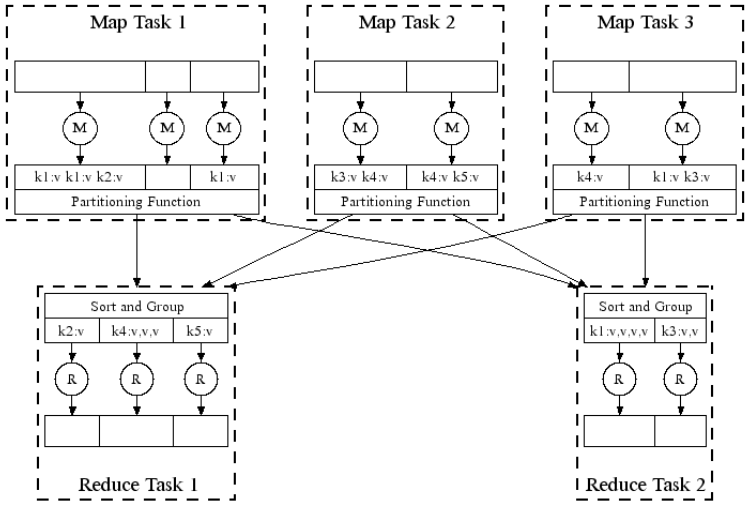


Refinements: Backup Tasks

- **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

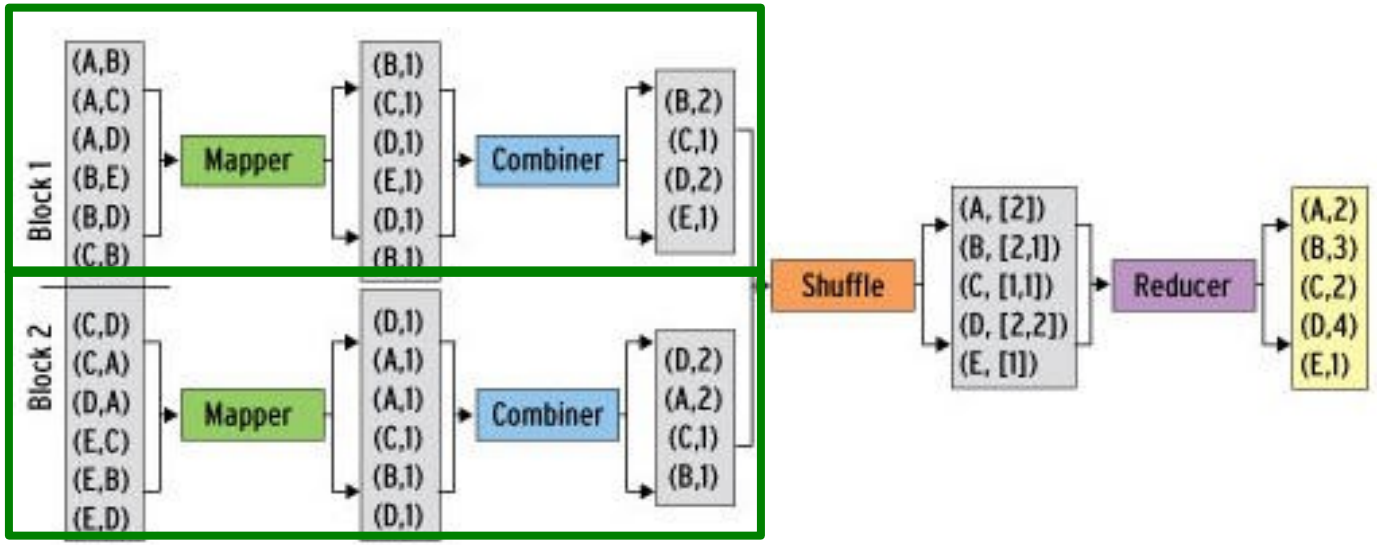
Refinement: Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ 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 (2)

- **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!
  - **NB:** It is still necessary to group, aggregate, and pass the result to the Reduce tasks, since there will typically be one key-value pair with a give key w coming from each of the Map tasks.

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

Problems Suited for Map-Reduce

Algorithms using Map-Reduce

- **MapReduce is not a solution to every problem**
- The entire distributed-file-system makes sense only for very large files that are rarely updated in place.
- DFS or MapReduce are not suitable for online sales, even if thousands compute nodes process Web requests.
- In fact, operations on online retailers data involve responding to searches, recording sales, and so on, which involve little calculation and might change the DB.
- They might still use MapReduce to perform analytic queries on big data, such as finding users whose buying patterns are most similar to a given user.

Map-Reduce for Google

- The original purpose for which the Google implementation of MapReduce was created was to execute very large matrix-vector multiplications as are needed in the calculation of PageRank
- Another important class of operations that can use MapReduce effectively are the relational-algebra operations.

Matrix-vector Multiplication by MapReduce

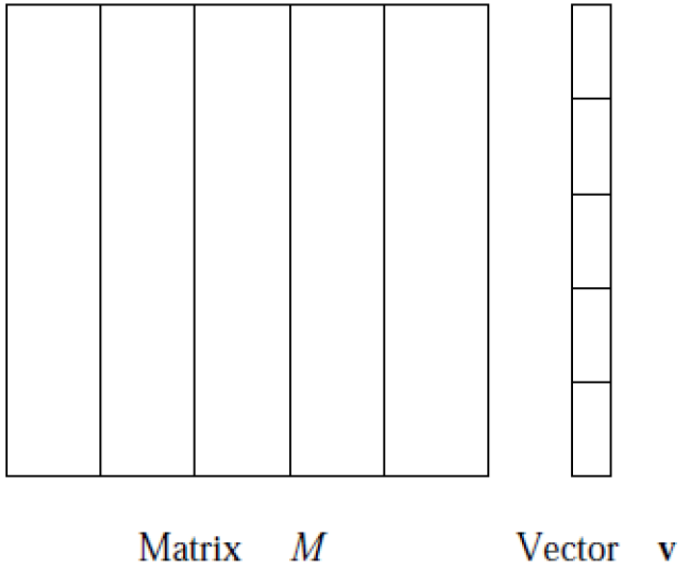
- Let M be an n × n matrix, with m_{ij} the element in row *i* and column *j*, and V be a vector of length n, whose j_{th} element is v_j
- The matrix-vector product is the vector x of length n, whose i_{th} element x_i is given by
  - $x_i = \sum_{j=1}^{n} m_{ij} v_j$
- If n = 100, we do not use a DFS or MapReduce, but for the ranking of Web pages in search engines n is in tens of billions.
- Let us assume n is large, but the vector V fits in main memory, hence be available to every Map task.
- M and V will be stored in a file of DFS. We assume that the coordinates of m_{ij} will be discoverable, either from its position in the file, or because it is stored with explicit coordinates (i, j, m_{ij}). Similar considerations for the position of element v_j

Matrix-vector Multiplication: Map and Reduce

- **Map Function:**
  - Each Map task operates on a chunk of M. From each element m_{ij} it produces the key-value pair (i, m_{ij}, v_j). Thus, all terms of the sum that make up the component x_i of the matrix-vector product will get the same key *i*.
- **Reduce Function**:
  - The Reduce function sums all the values associated with a given key i. The result will be a pair (i, x_i).

What if vector does not fit main memory

- the matrix is divided into vertical stripes of equal width
- the vector is divided into an equal number of horizontal stripes, of the same height so that each fits a computing node's memory



- ith stripe of the matrix multiplies ith stripe of the vector.
- Each Map task is assigned a chunk from one stripes of the matrix and the corresponding stripe of the vector.
- Then, Map and Reduce tasks work as the previous case.

Example: Host size

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host
- **Other examples:**
  - Link analysis and graph processing
  - Machine Learning algorithms

Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

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

⋈

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)*.

Cost Measures for Algorithms

- **In MapReduce we quantify the cost of an algorithm using**

1. *Communication cost* = total I/O of all processes (number of bytes or tuples)
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but counting only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost =** input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes (each file passed from map to reduce processes is counted twice, since it is first sent for shuffling and then to reduce tasks).
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process.

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other
  - Communication cost tend to dominate in Map/Reduce

- Elapsed cost is wall-clock time using parallelism

Communication vs Processing Cost

- Processing costs rarely dominate communication costs:
  - The algorithm executed by each task tends to be simple, often linear in the size of its input
  - Even 1Gbit/sec of cluster interconnection speed is slow compared with CPU's instruction execution speed
  - Since there is competition for the interconnect when several compute nodes communicate simoultaneously, they can do a lot of work on a received input element during the time to deliver it
  - Even if a task executing at a compute node has a copy of the chunk(s) it processes, the chunk normally will be stored on disk, and the time to move data into main memory may exceed the time to operate on it once it is in memory

Cost of Map-Reduce Join

- **Total communication cost**
  $= O(|R|+|S|+|R \bowtie S|)$
- **Elapsed communication cost** = O(s)
  - We're going to pick ***k*** and the number of Map processes so that the I/O limit ***s*** is respected
  - We put a limit ***s*** on the amount of input or output that any one process can have. ***s* could be:**
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like communication cost

Pointers and Further Reading

Implementations

- Google
  - Not available outside Google
- **Hadoop**
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce

Cloud Computing

- Ability to rent computing by the hour
  - Additional services e.g., persistent storage
- Amazon's "Elastic Compute Cloud" (EC2)
- Aster Data and Hadoop can both be run on EC2
- **For CS341 (offered next quarter) Amazon will provide free access for the class**

Reading

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters
  - http://labs.google.com/papers/mapreduce.html

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
  - http://labs.google.com/papers/gfs.html

Resources

- Hadoop Wiki
  - Introduction
    - http://wiki.apache.org/lucene-hadoop/
  - Getting Started
    - http://wiki.apache.org/lucene-hadoop/GettingStartedWithHadoop
  - Map/Reduce Overview
    - http://wiki.apache.org/lucene-hadoop/HadoopMapReduce
    - http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses
  - Eclipse Environment
    - http://wiki.apache.org/lucene-hadoop/EclipseEnvironment
- Javadoc
  - http://lucene.apache.org/hadoop/docs/api/

Resources

- Releases from Apache download mirrors
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- Nightly builds of source
  - http://people.apache.org/dist/lucene/hadoop/nightly/
- Source code from subversion
  - http://lucene.apache.org/hadoop/version_control.html

Further Reading

- Programming model inspired by functional language primitives
- Partitioning/shuffling similar to many large-scale sorting systems
  - NOW-Sort ['97]
- Re-execution for fault tolerance
  - BAD-FS ['04] and TACC ['97]
- Locality optimization has parallels with Active Disks/Diamond work
  - Active Disks ['01], Diamond ['04]
- Backup tasks similar to Eager Scheduling in Charlotte system
  - Charlotte ['96]
- Dynamic load balancing solves similar problem as River's distributed queues
  - River ['99]