DATA MINING LECTURE 15

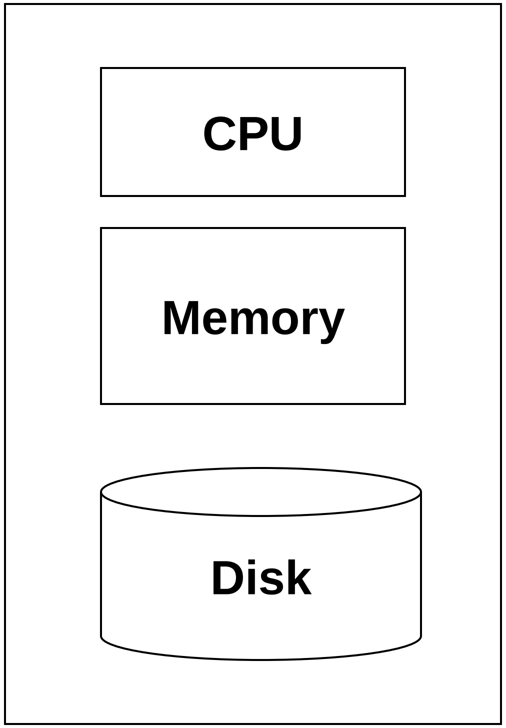
The Map-Reduce Computational Paradigm

Most of the slides are taken from:
Mining of Massive Datasets
Jure Leskovec, Anand Rajaraman, Jeff Ullman
Stanford University
http://www.mmds.org

Large Scale data mining

- **Challenges:**
  - How to deal with massive amount of data?
    - Storing the web requires Petabytes of data!
  - 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



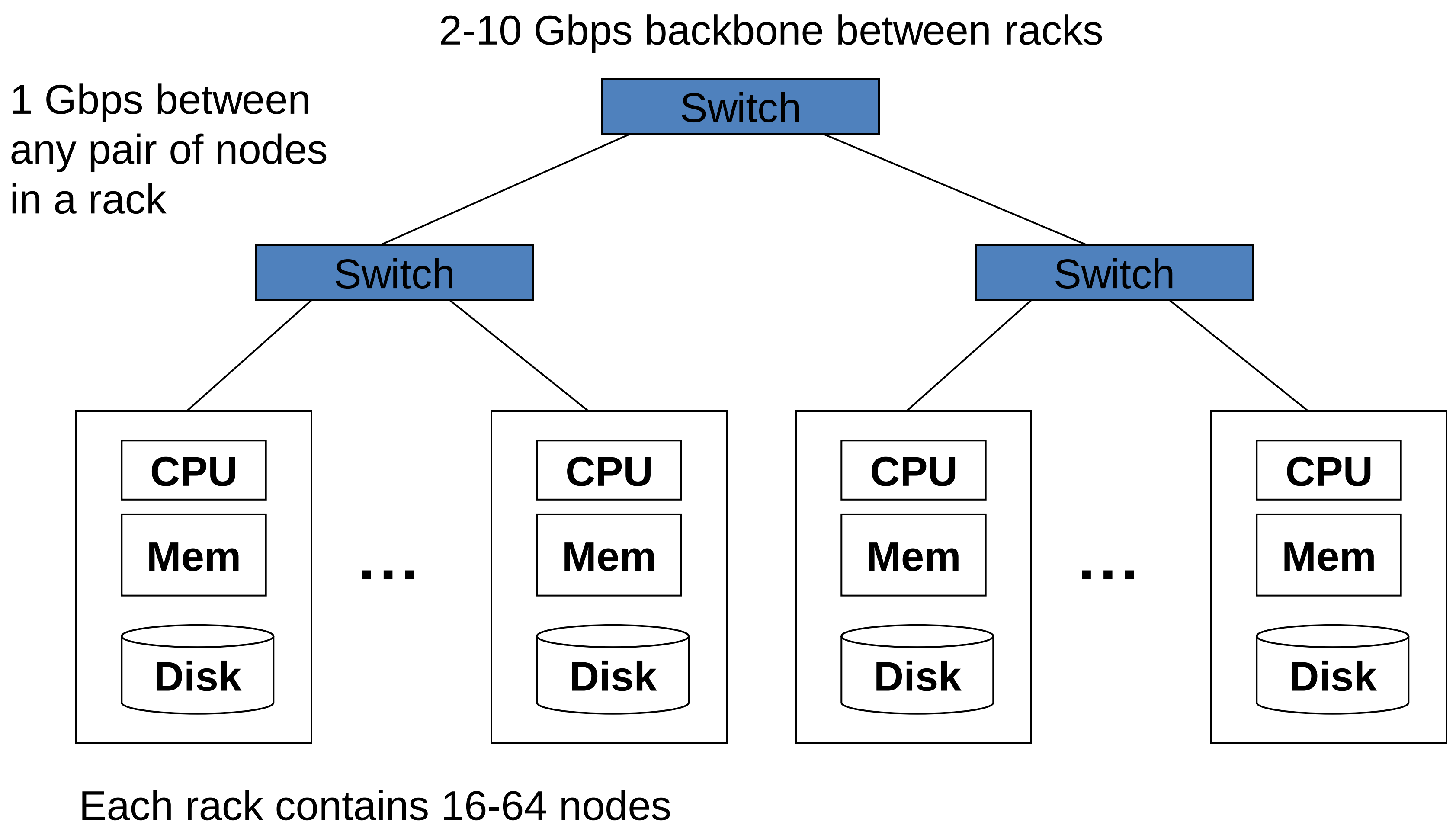
Machine Learning, Statistics

"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



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
  - **Programming model**
    - Map-Reduce

Storage Infrastructure

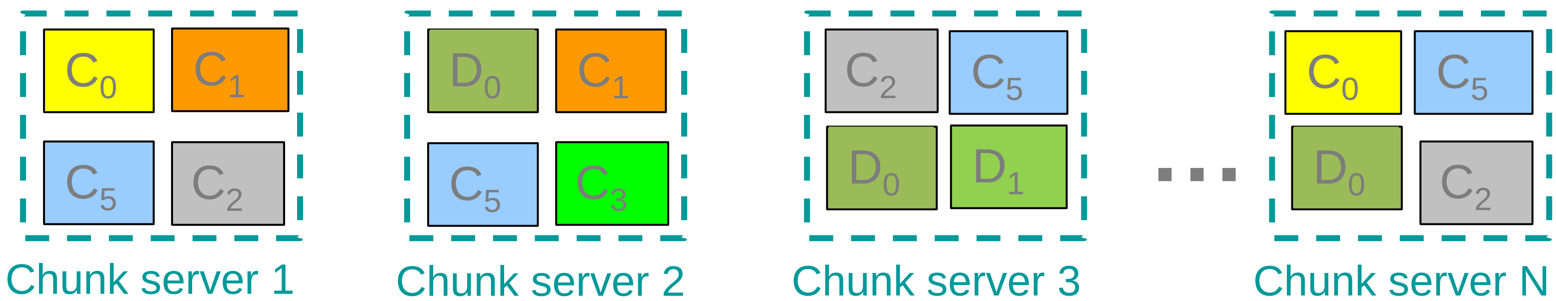
- **Problem:**
  - If nodes fail, how to store data persistently?
- **Answer:**
  - **Distributed File System:**
    - Provides global file namespace
    - Google GFS; Hadoop HDFS;
- **Typical usage pattern**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends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

Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 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 also 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

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
  - Find the frequency of words in the Web.

Task: Word Count

Case 1:

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

Case 2:

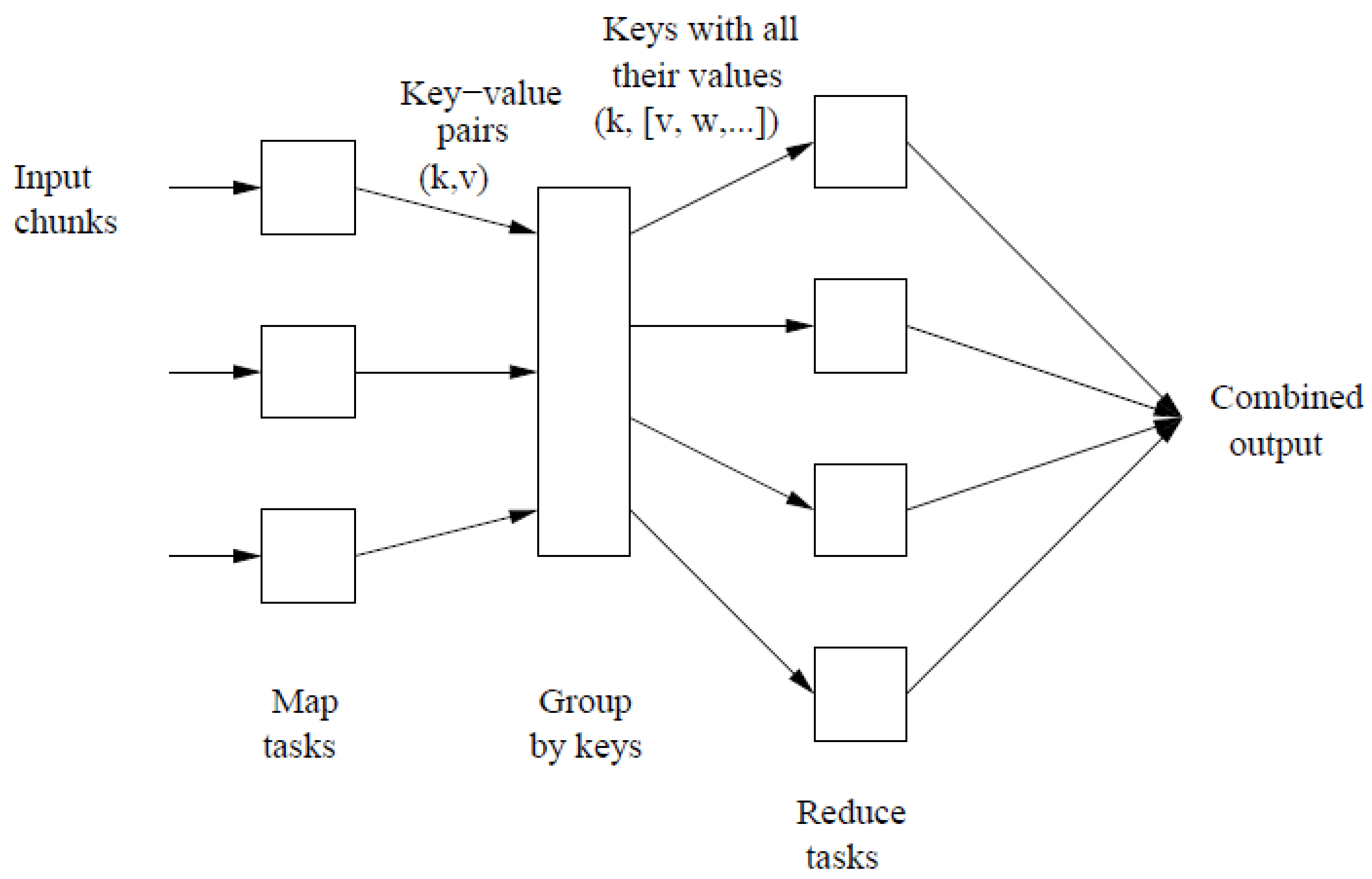
- Count occurrences of words:
  - **`words(doc.txt) | sort | uniq -c`**
    - where **`words`** takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable

MapReduce: Overview

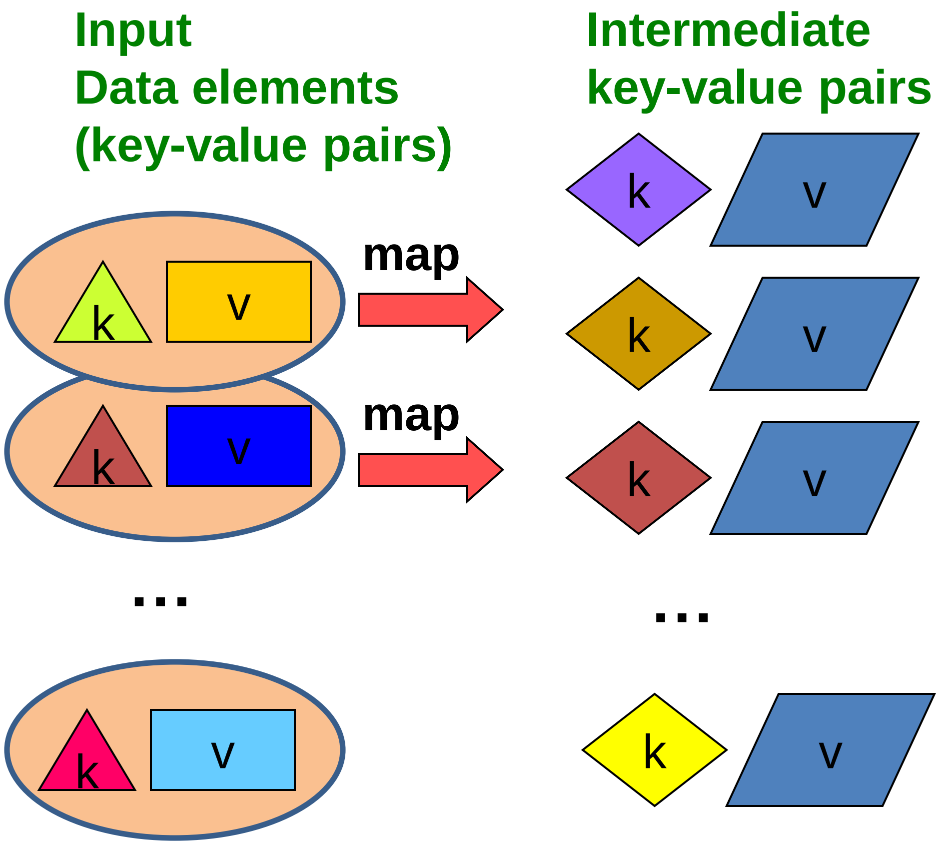
- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

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

MapReduce in a figure

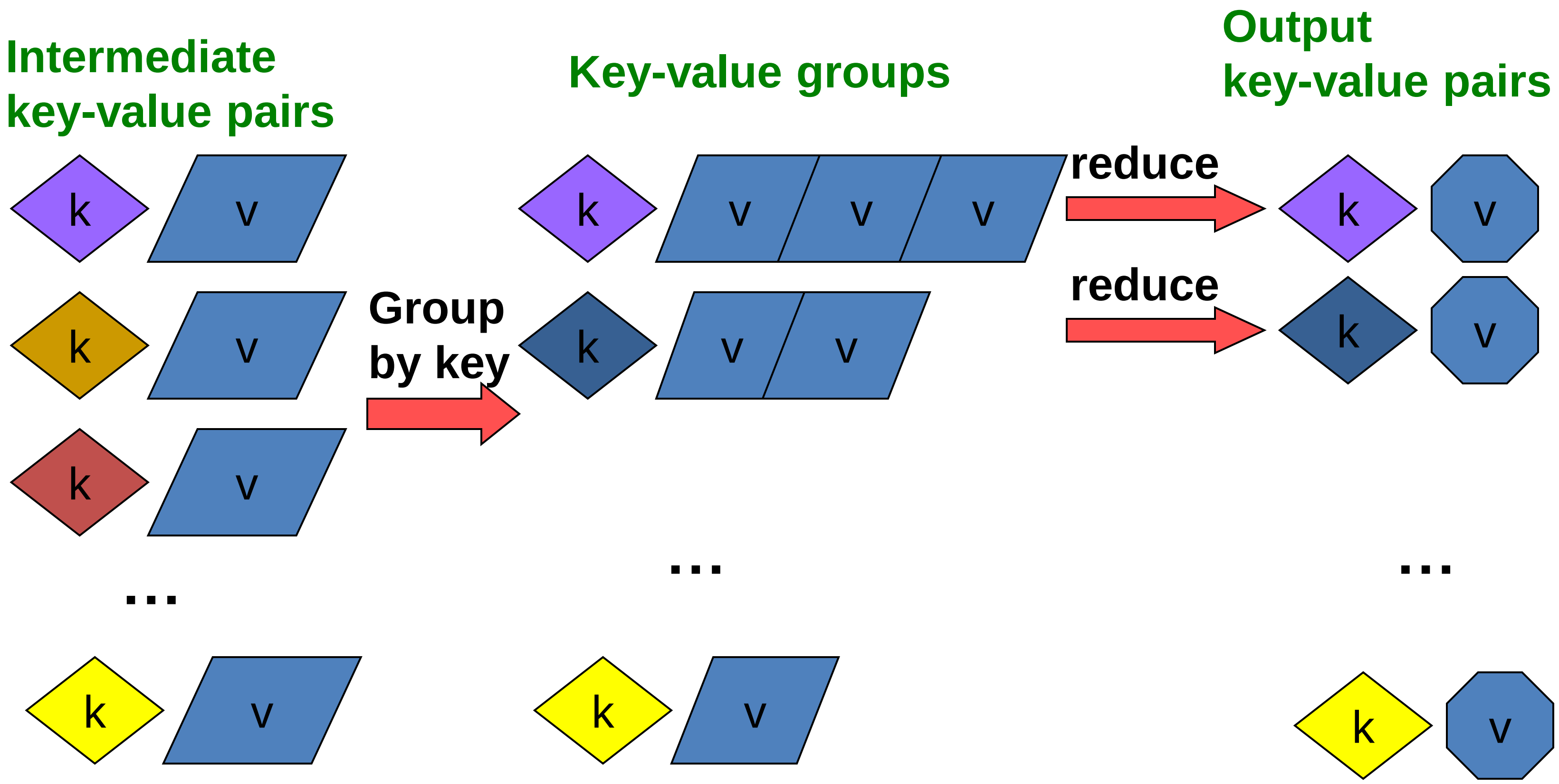


MapReduce: The Map Step



Important:
Different shapes correspond to different types of keys and values!

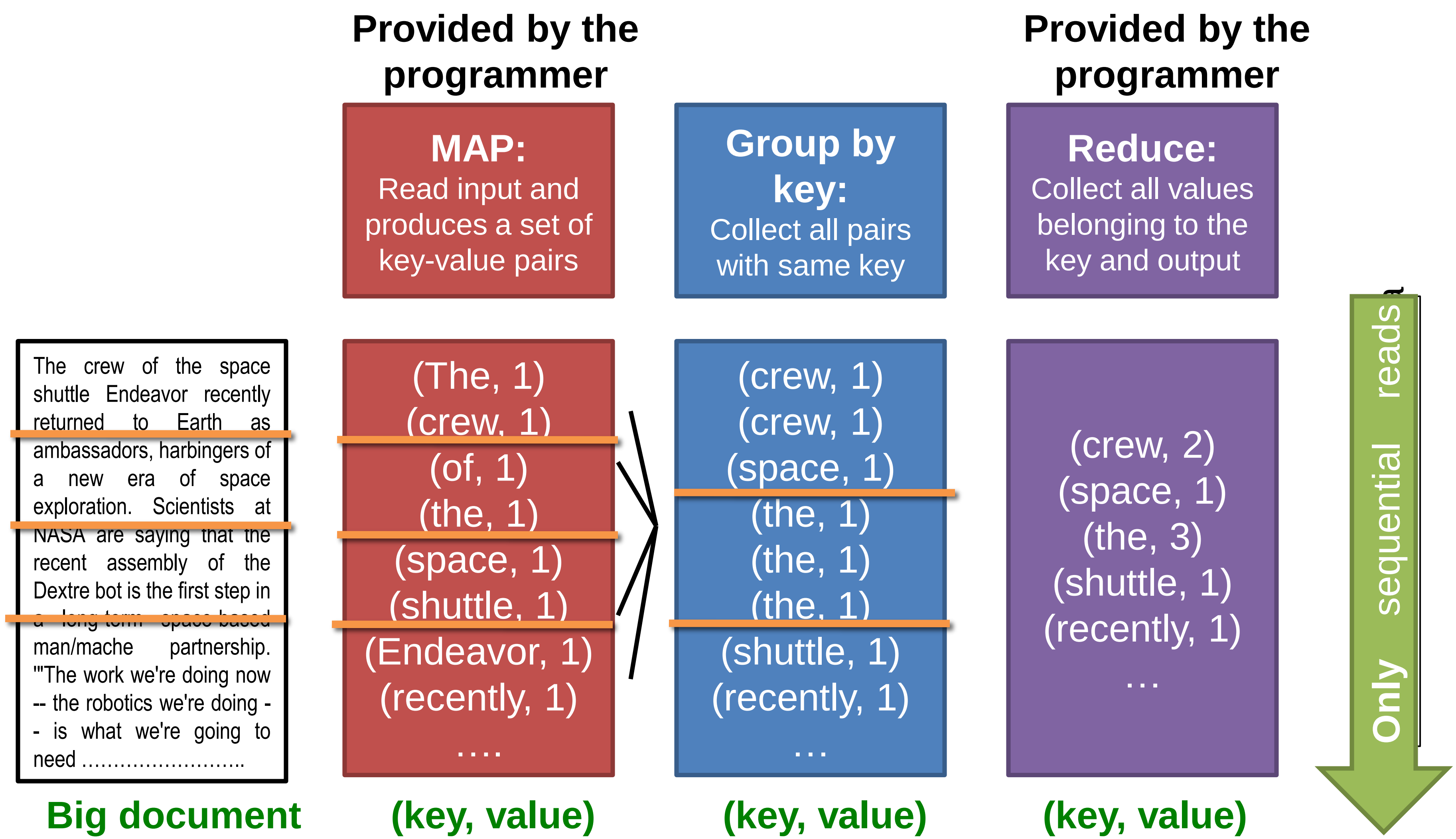
MapReduce: The Reduce Step



More Specifically

- Input: a set of data elements that we think of as key-value pairs
  - E.g., key is the filename, value is a single line in the file
- Programmer specifies two methods:
  - **Map(k, v)** $\rightarrow$ <k', v'>*
    - Takes a key-value pair and outputs a set of new key-value pairs
      - E.g., the key k' is a word and the value v' is 1. One such pair is produced for each appearance of the word in the input line
    - 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'
    - The output is a new key value pair, where for each key k' a new value v'' is computed from the set of values associated with k'
      - E.g., the value v'' is the sum of values v'

MapReduce: Word Counting



Word Count Using MapReduce

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

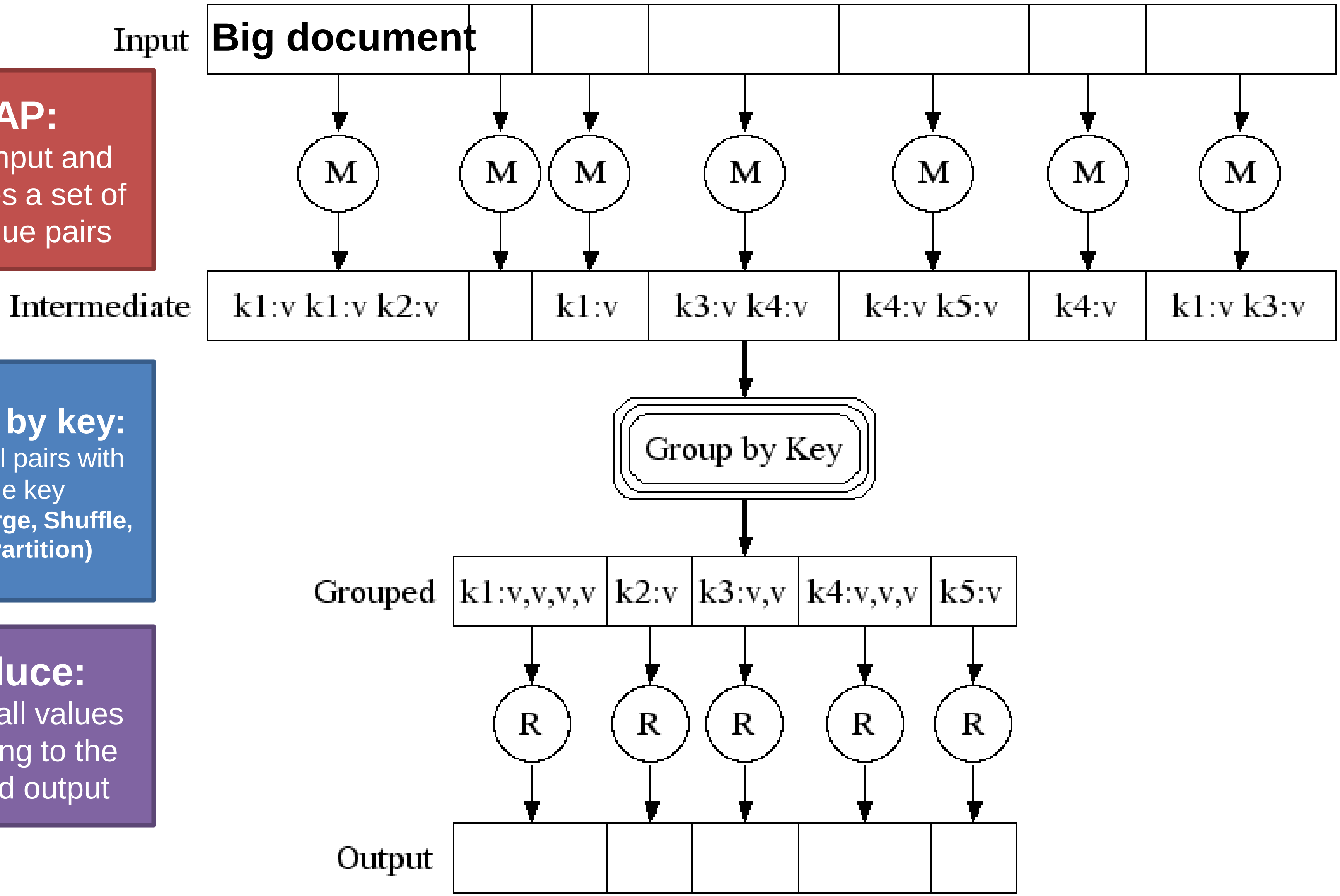
- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the group by key step
- Handling machine failures
- Managing required inter-machine communication

Map-Reduce: A diagram

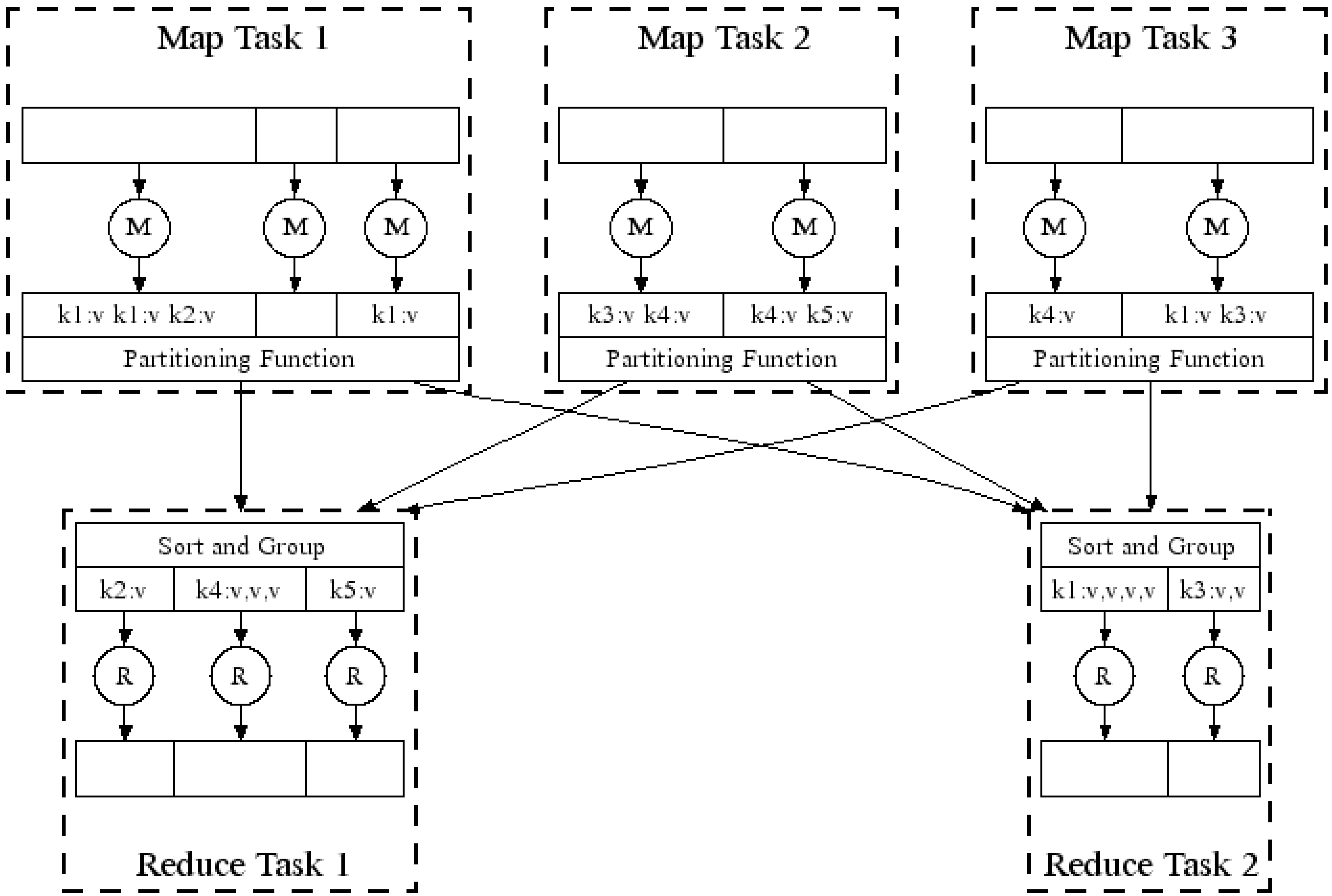
MAP:
Read input and produces a set of key-value pairs

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

Reduce:
Collect all values belonging to the key and output



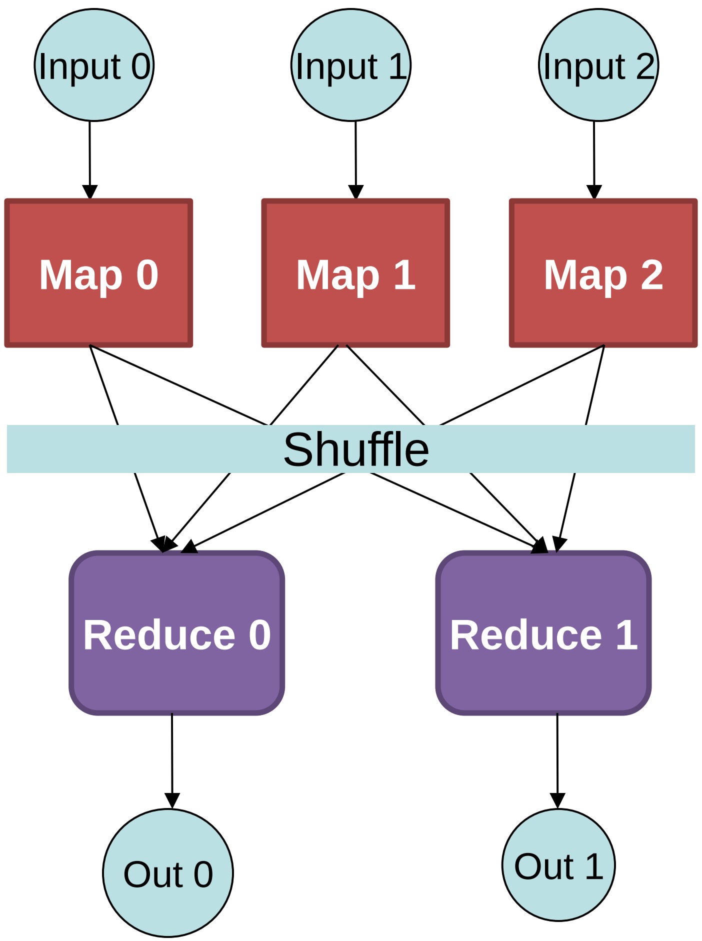
Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

Map-Reduce

- Programmer specifies:
  - **Map** and **Reduce** and input files
- **Workflow:**
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input (k,v)-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



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

Overview

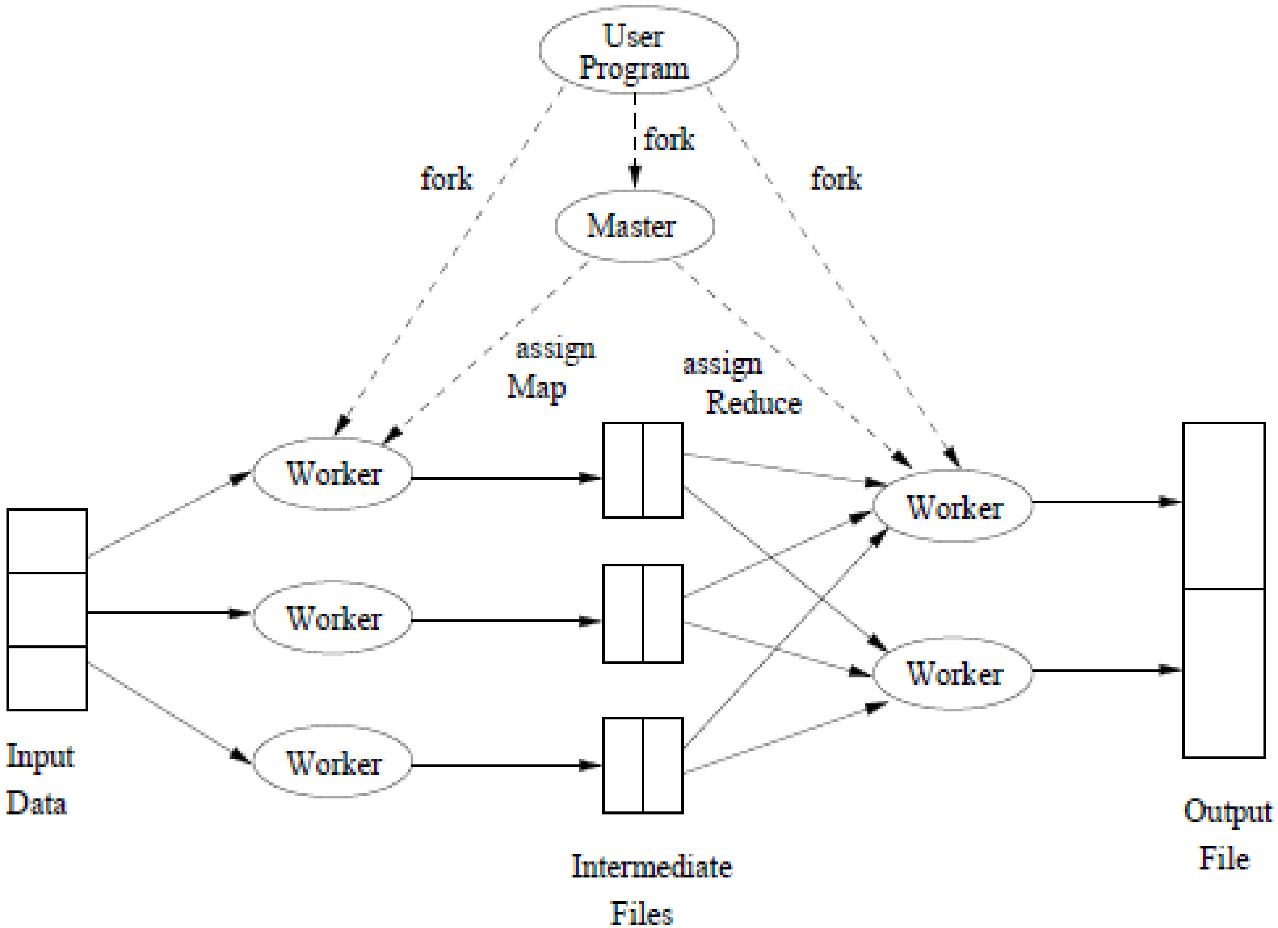


Figure 2.3: Overview of the execution of a MapReduce program

Dealing with Failures

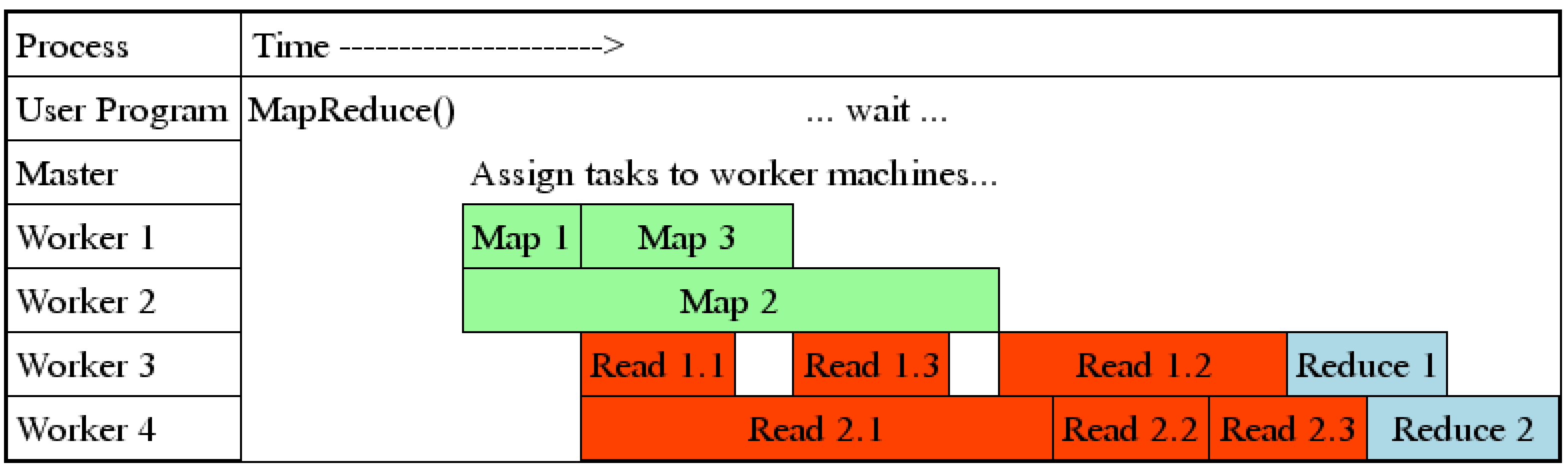
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce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

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

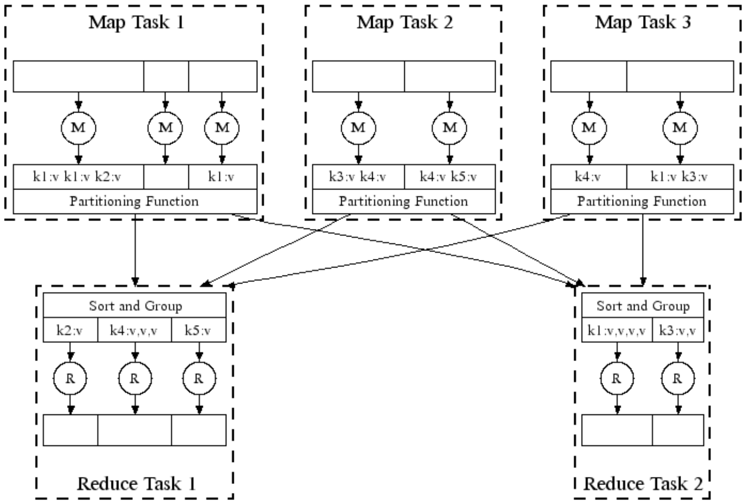


Refinements: Backup Tasks

- **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

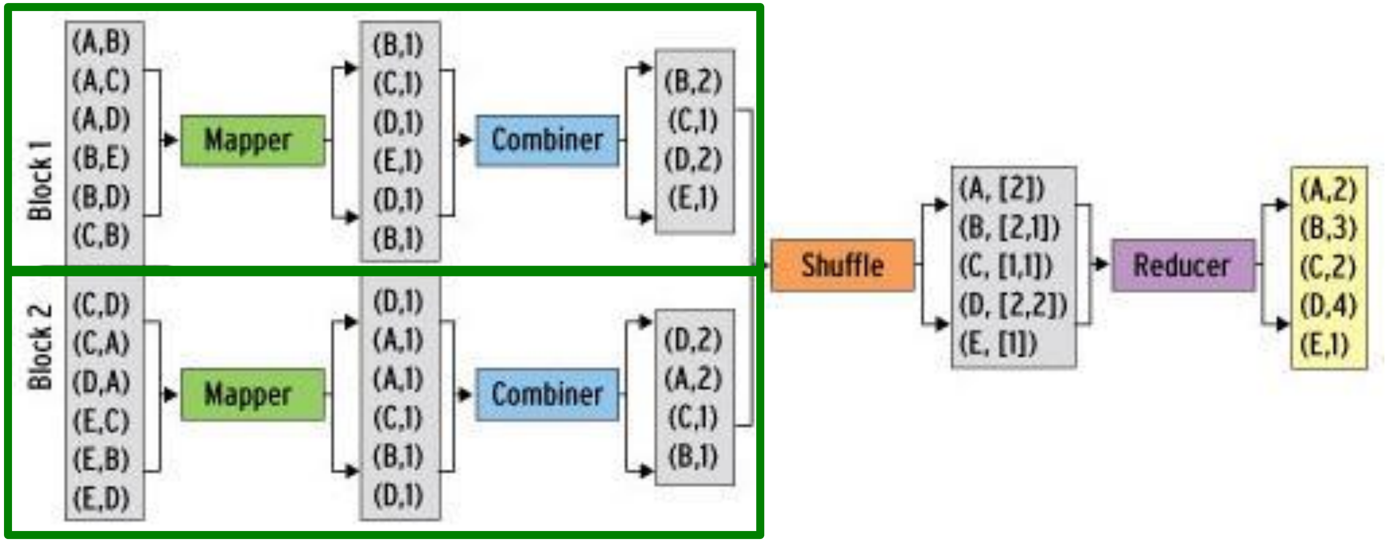
- Often a **Map** task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key k
  - E.g., popular words in the word count example
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function



- Works only if **Reduce** function is commutative and associative

Refinement: Combiners

- **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

PROBLEMS SUITED FOR MAP-REDUCE

Examples

- **Counting tasks**
  - Find the total size in bytes of a host
  - Compute the frequency of all k-grams on the web
  - Compute the frequency of queries
  - Compute the frequency of query,url pairs
- **Other examples:**
  - Link analysis and graph processing – PageRank
  - Machine Learning algorithms
  - Linear algebra operations (matrix-vector, matrix-matrix multiplication)

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	B	C
a_3	b_2	c_1
a_3	b_2	c_2
a_4	b_3	c_3

Map-Reduce Join

- **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)* (where h is a hash function)
  - Hadoop does this automatically; just tell it what the key is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* from the list of values associated with b, and outputs *(a,b,c)*.

Other database operations

- All SQL operations can be implemented using map-reduce:
  - Select
  - Project
  - Union
  - Difference
  - Equi-Join
  - Left-outer join

Matrix-Vector multiplication

- Compute the product of matrix M with vector v

$$(Mv)_i = \sum_j m_{ij} v_j$$

- This is an operation that appears very often in many different tasks
  - E.g., the computation of the PageRank vectors.
  - The size of the Web matrix is in the order of billions! But it is a very sparse matrix

- Storage:

The matrix and vectors are stored in a sparse form:

  - Triplets of the form (i, j, m_{ij}) for the non-zero entries of the matrix
  - Pairs of the form (i, v_i) for the elements of the vector.

Matrix-vector multiplication

- Case 1: The vector fits in memory
  - In this case the vector that we want multiply is loaded in memory at each mapper.

- Recall that we want to compute:

$$\sum_j m_{ij} v_j$$

for entry i of the output vector.

- How should we define the map-reduce process?
  - The **mapper** reads a chunk of the matrix M, and for each entry (i, j, m_{ij}) it outputs the key-value pair $(i, m_{ij} v_j)$
  - The **reducer** takes the sum of all values that are associated with row i.

Matrix-vector multiplication

- Case 2: The vector does not fit in memory
- In this case we split the matrix and the vector into stripes:

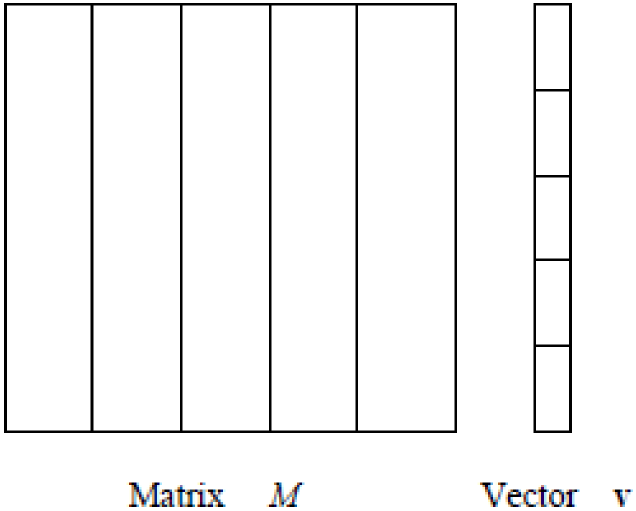


Figure 2.4: Division of a matrix and vector into five stripes

- We perform the computation for each stripe of the matrix, where the vector can fit into memory
  - For PageRank it is better to split the matrix into blocks.

Extenstions: Pregel- Giraph

- Data and computation is modeled as a Graph.
  - Each node in the graph handles a task
  - Each node output messages to the remaining nodes
  - Each node processes the incoming messages from other nodes.

- Computation is performed in supersteps:
  - In one superstep all messages are processed, and new messages are sent out.

- Failures
  - The computation is periodically checkpointed after a number of supersteps.

- Pregel: developed by Google. Giraph: open-source version
  - Although a general computation model, it is usually used for computations on graphs.

Example: All pairs shortest paths

- Data: the edges of a large graph with weights
- Compute: the shortest path between any two nodes

- Each node in Pregel stores information about a node in the input graph and connects with its neighbors
  - For node a we store the pairs (b, w_{ab}) with the distance of a to all other nodes
    - Initially only to immediate neighbors
  - At each step each node a broadcasts the distances (a, b, w_{ab}) to its neighbors.
  - When node a receives message (c, d, w_{cd}), it checks if there are pairs (c, w_{ac}) and (d, w_{ad}) stored locally
  - If $w_{ac} + w_{cd} < w_{ad}$ then it updates the pair (d, w_{ad}).

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

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
- Hadoop releases from Apache download mirrors
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- Javadoc
  - http://lucene.apache.org/hadoop/docs/api/

Other systems

- Apache Spark
  - https://spark.apache.org/
  - A different distributed computation software stack running over HDFS, or Amazon S3
  - Developed by UC Berkeley

- On top of Apache Spark:
  - Spark SQL: allows for querying structured and semi-structured data
  - MLlib – Apache Mahout: Distributed Machine Learning framework
    - Implements clustering, classification, dimensionality reduction algorithims
  - GraphX: Distributed Graph processing framework, similar to Pregel
    - Implements several graph processing algorithms

Other systems

- Apache Hive:
  - https://hive.apache.org/
  - Distributed Data Warehousing system. Works over HDFS and Amazon S3.
  - HiveQL: SQL like querying language.
  - Developed by Facebook.
- GraphLab and GraphChi
  - Distributed Graph processing framework
  - Pregel-like computation

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
- R on the Cloud:
  - Several resources that allow to run R scripts on the cloud. Useful for bio-informatics applications.