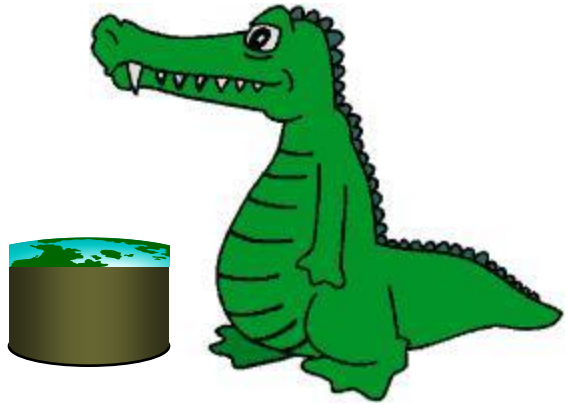
CAP4770/5771
Introduction to Data Science
Fall 2015

University of Florida, CISE Department
Prof. Daisy Zhe Wang



Based on notes from CS194 at UC Berkeley by Michael Franklin, John Canny, and Jeff Hammerbacher

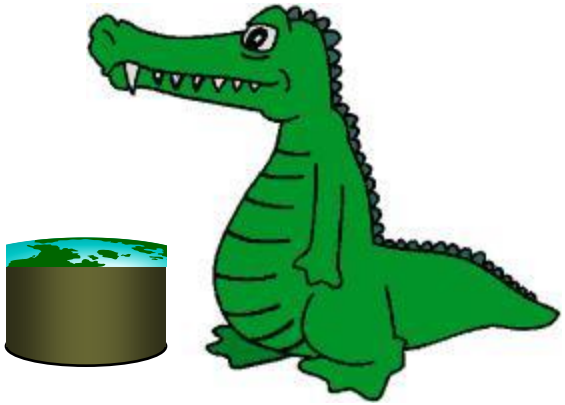
Map/Reduce: Simplified Data Processing on Large Clusters

Parallel/Distributed Computing

Distributed File System

M/R Programming Model

Parallel Analytics using M/R



Adapted Slides from Jeff Ullman, Anand Rajaraman and Jure Leskovec from Stanford

Logistics

- Finish all the AWS tutorials (AWS account, EC2, S3, EMR), setup and test run word-count M/R job by this Friday
- Lab 3:
  - Group of 2 due Friday
  - Out this weekend, due next Friday
  - Start early!!
- Midterm – the week after the next
- Pop quiz coming up

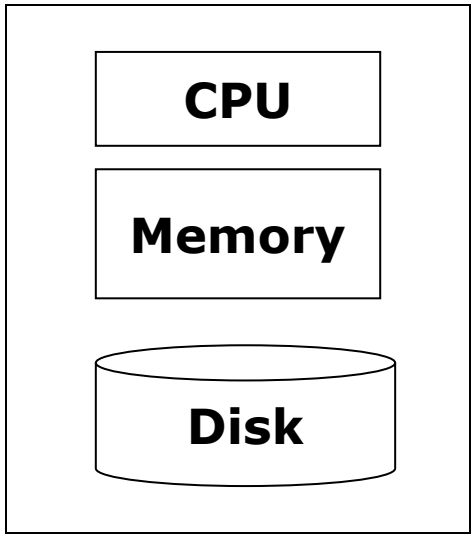
Parallel Computing

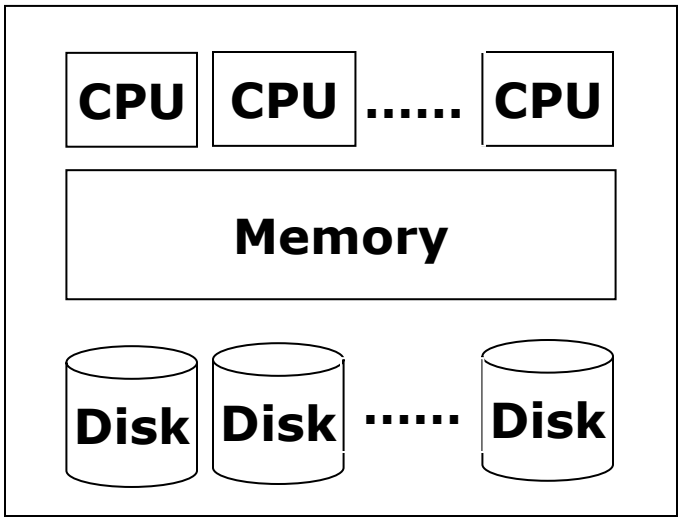
- MapReduce is designed for parallel computing
- Before MapReduce
  - Enterprise: a few super-computers, parallelism is achieved by parallel DBs (e.g., Teradata)
  - Science, HPC: MPI, openMP over commodity computer clusters or over super-computers for parallel computing, do not handle failures (i.e., Fault tolerance)
- Today:
  - Enterprise: map-reduce, parallel DBs
  - Science, HPC: openMP, MPI and map-reduce

MapReduce can apply over both multicore and **distributed environment**

Data Analysis on Single Server

- Many data sets can be very large
  - Tens to hundreds of terabytes
- Cannot always perform data analysis on large datasets on a single server (why?)
  - Run time (i.e., CPU), memory, disk space





Motivation: Google's Problem

- 20+ billion web pages * 20KB = 400+ TB
  - ~1000 hard drives to store the web
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web

- Takes even more to do something useful with the data!
  - CPU
  - Memory

Distributed Computing

- Challenges:
  - Distributed/parallel programming is hard
    - How to distribute Data? Computation? Scheduling? Fault Tolerant? Debugging?
  - On the software side: what is the programming model?
  - On the hardware side: how to deal with hardware failures?
- MapReduce/Hadoop addresses all of the above
  - Google's computational/data manipulation model
  - Elegant way to work with big data

Commodity Clusters

- Recently standard commodity architecture for such problems:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnect
- Challenge: How to organize computations on this architecture?
  - handle issues such as hardware failure

Stable storage

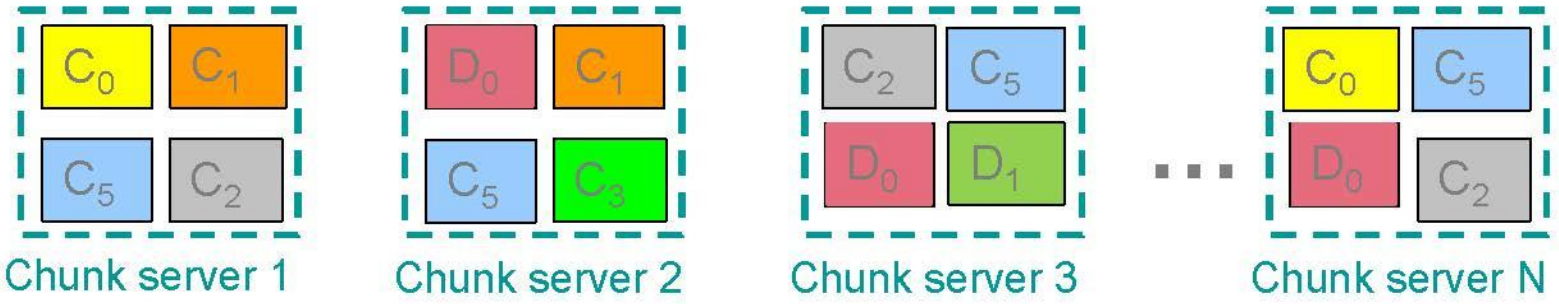
- First order problem: if nodes can fail, how can we store data persistently?
- Idea:
  - Store files multiple times for reliability
- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely 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 Nodes in HDFS
  - Stores metadata
  - Likely to be replicated as well
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

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 Motivation

- Large-Scale Data Processing
  - Want to use 1000s of CPUs, But don't want the hassle of scheduling, synchronization, etc.
- MapReduce provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling and synchronization
  - Monitoring & status updates

MapReduce Basics

- MapReduce
  - Programming model similar to Lisp
  - (and other functional languages)
- Many problems can be phrased using Map and Reduce functions
- Benefits for implementing an algorithm in MapReduce API
  - Automatic parallelization
  - Easy to distribute across nodes
  - Nice retry/failure semantics

Example Problem: Word Count

- We have a huge text file: doc.txt or a large corpus of documents: docs/*
- Count the number of times each distinct word appears in the file.. Apps?
- Sample application: analyze web server logs to find popular URLs

Word Count (2)

- Case 1: File too large for memory, but all <word, count> pairs fit in memory
- Case 2: File on disk, too many distinct words to fit in memory
  - Count occurrences of words

```
words(doc.txt) | sort | uniq -c
```

  - `words` takes a file and output the words in it, one word a line
  - Naturally capture the essence of MapReduce and naturally parallelizable

3 steps of MapReduce

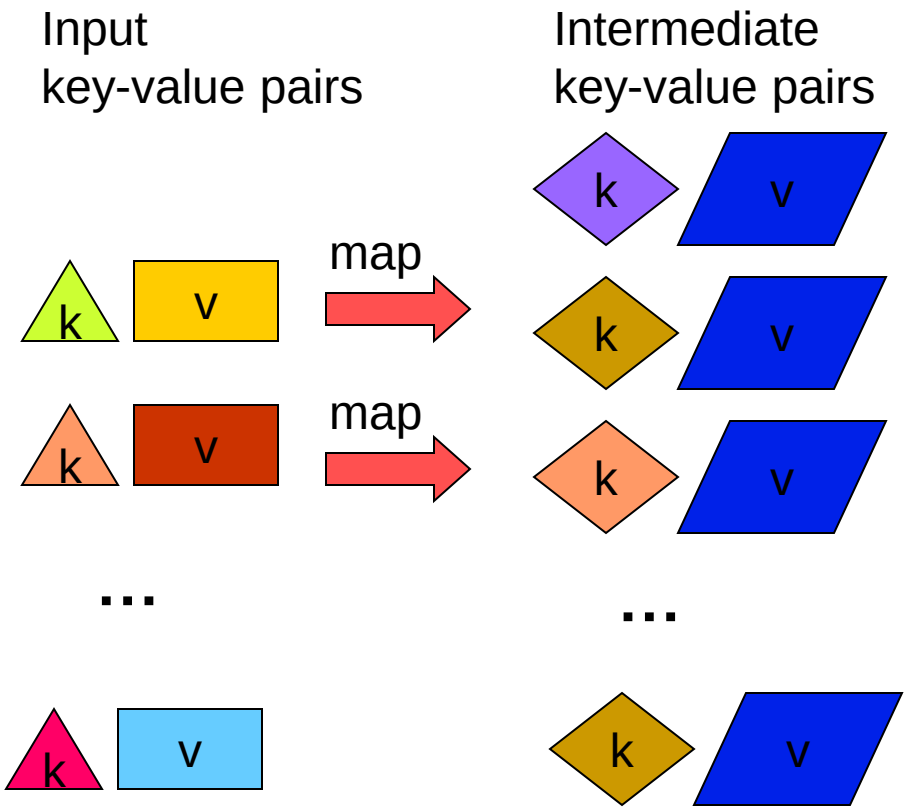
- Sequentially read a lot of data
- **Map:** Extract something you care about
- **Group by key:** Sort and shuffle
- **Reduce:** Aggregate, summarize, filter or transform
- Output the result

For different problems, steps stay the same,
Map and Reduce change to fit the problem

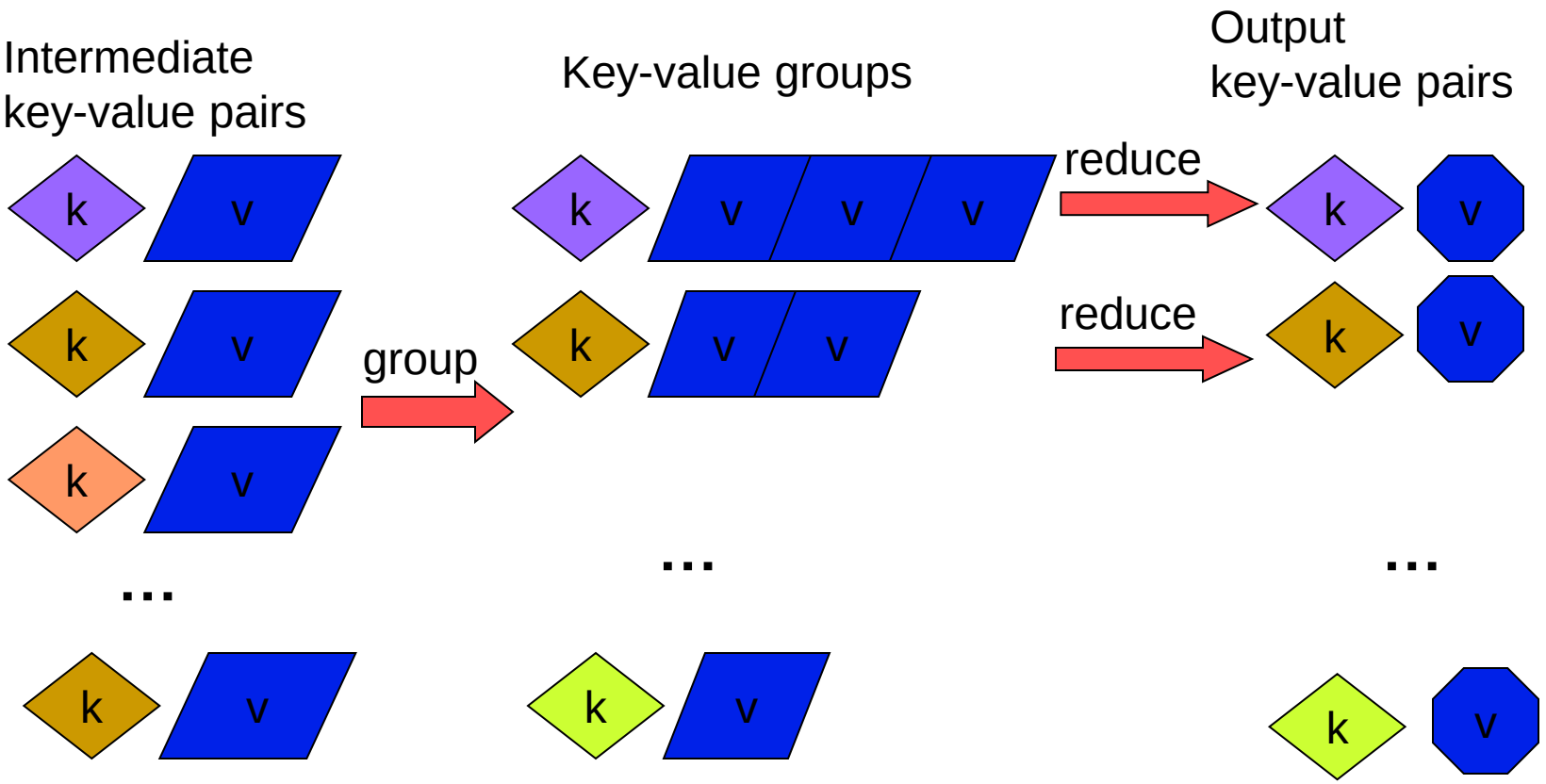
MapReduce: more detail

- Input: a set of key/value pairs
- Programmer specifies two methods
  - Map(k,v) → list(k1,v1)
    - Takes a key-value pair and outputs a set of key-value pairs (e.g., key is the file name, value is a single line in the file)
    - One map call for every (k,v) pair
  - Reduce(k1, list(v1)) → list (k1, v2)
    - All values v1 with the same key k1 are reduced together to produce a single value v2
    - There is one reduce call per unique key k1
- Group by key is executed by default in the same way: Sort(list(k1,v1)) → list (k1, list(v1))

MapReduce: The Map Step



MapReduce: The Group and Reduce Step

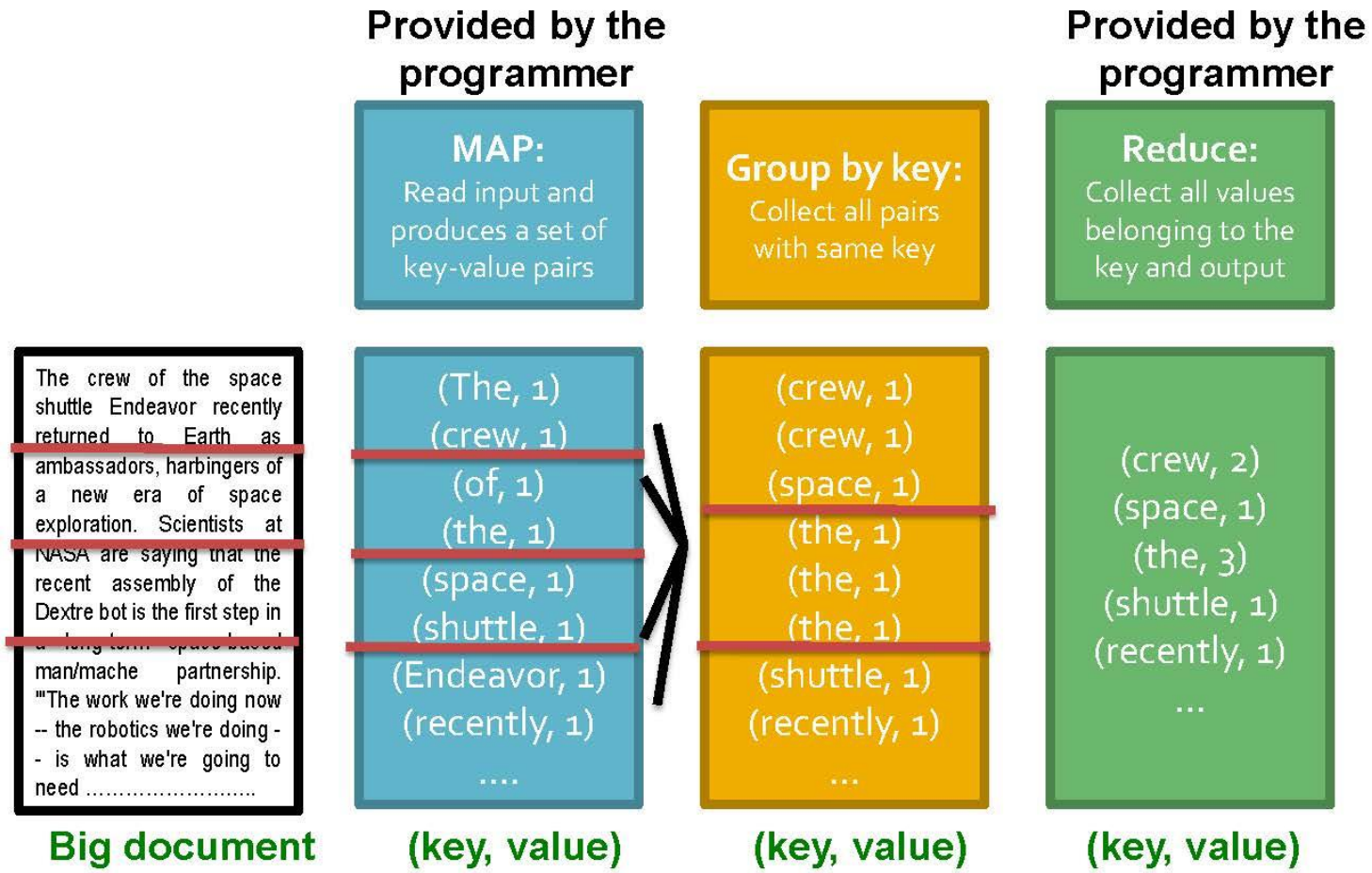


Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of 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(result)
```

MapReduce: Word Count

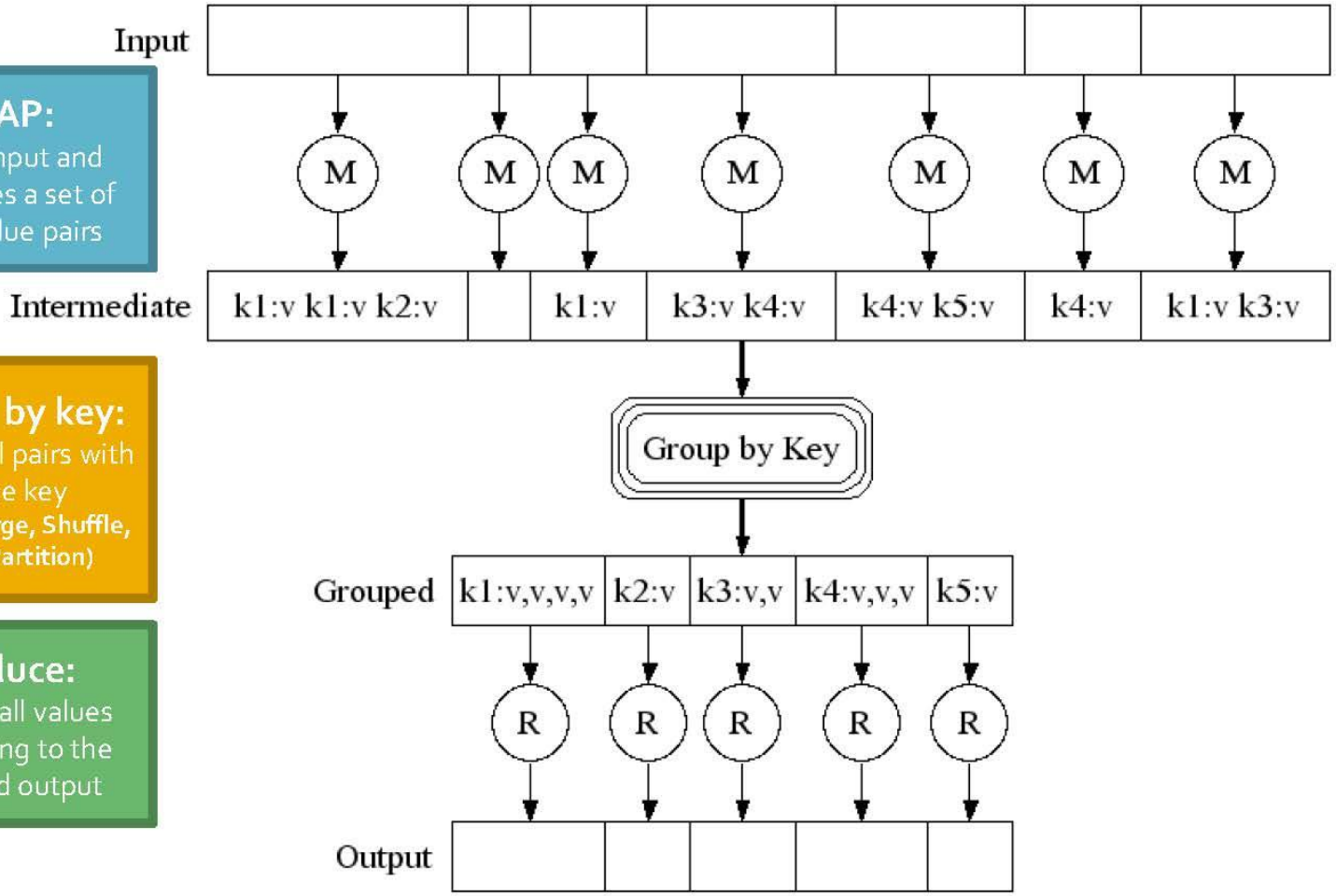


MapReduce Execution Details

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



Combiners

- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
  - combine(k1, list(v1)) → v2
  - Usually same as reduce function
- Works only if reduce function is commutative and associative

Implementations

- Google
  - Not available outside Google
- Hadoop
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/

Many other Analytics and Algorithms can be Parallelized using MapReduce

Exercise 1: Host size

- Suppose we have a large web corpus
- Let's look at the metadata file
  - Lines of the form (URL, size, date, …)
- For each host, find the total number of bytes
  - i.e., the sum of the page sizes for all URLs from that host

Example 1: Host size (cont.)

- InputFormats: transform on-disk data representation on HDFS to in-memory key-value
  - Example: TextInputFormat, KeyValueTextInputFormat

- Mapper: (position, "URL, size, data,…") -> (hostname, size)
- Mapper: (URL, "size, data,…") -> (hostname, size)
- Reducer: (hostname, list (size)) -> (hostname, totalsize)