Analyzing Massive Data Sets
Summer Semester 2019

Prof. Dr. Peter Fischer

Institut für Informatik

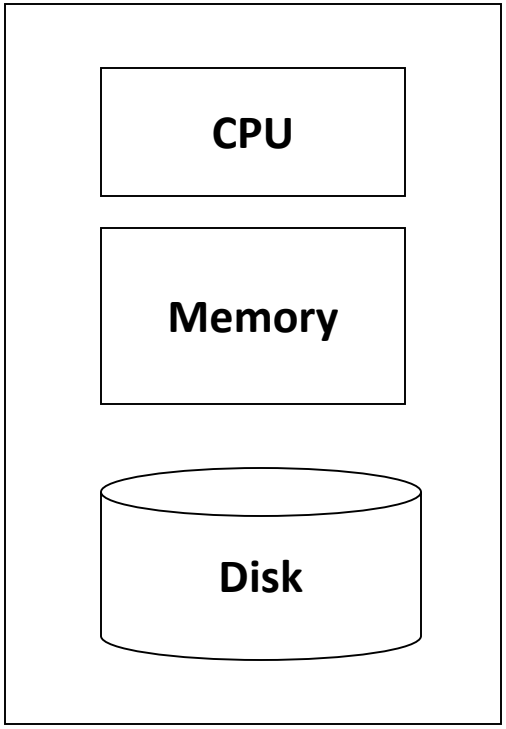
Lehrstuhl für Datenbanken und Informationssysteme

Chapter 2: HDFS & MapReduce

MapReduce

- Much of the course will be devoted to **large scale computing** for **data analysis**
- **Challenges:**
  - How to distribute computation?
  - How to distribute data?
  - How to deal with failures?
  - ➢ 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



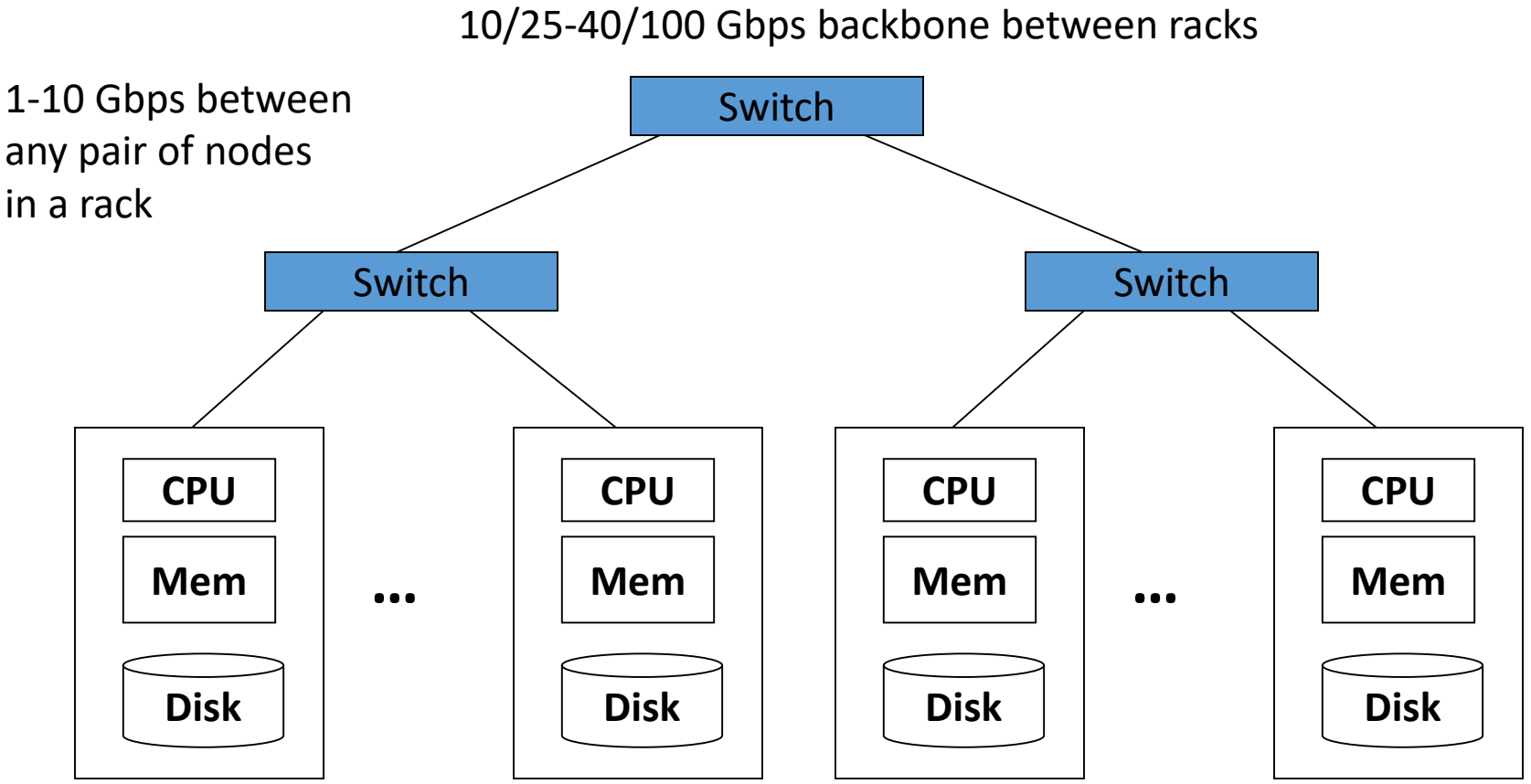
Machine Learning, Statistics

“Classical” Data Mining

Motivation: Google Example (early 2000s)

- 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, there is a standard architecture for such problems**
  - 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 analysis** problems on **commodity hardware**
- **Challenges:**
  - How do you distribute computation ?
  - How do you distribute data?
  - **How can we make it easy to write distributed programs?**
    - **How do we parallelize and synchronize?**
    - **How do we minimize communication?**
  - Machines fail!

Considerations on failures

- One server may stay up 3 years (1,000 days)
- If you have 1,000 servers, expect to lose 1/day
- People estimated Google had ~1M machines in 2011
  - 1,000 machines fail every day!
- **"Ultra-reliable" hardware doesn't really help**
  - At large scales, the most reliable hardware still fails, albeit less often
  - Software still needs to be fault-tolerant
- Commodity machines without fancy hardware give better **performance/$**

Idea and Solution

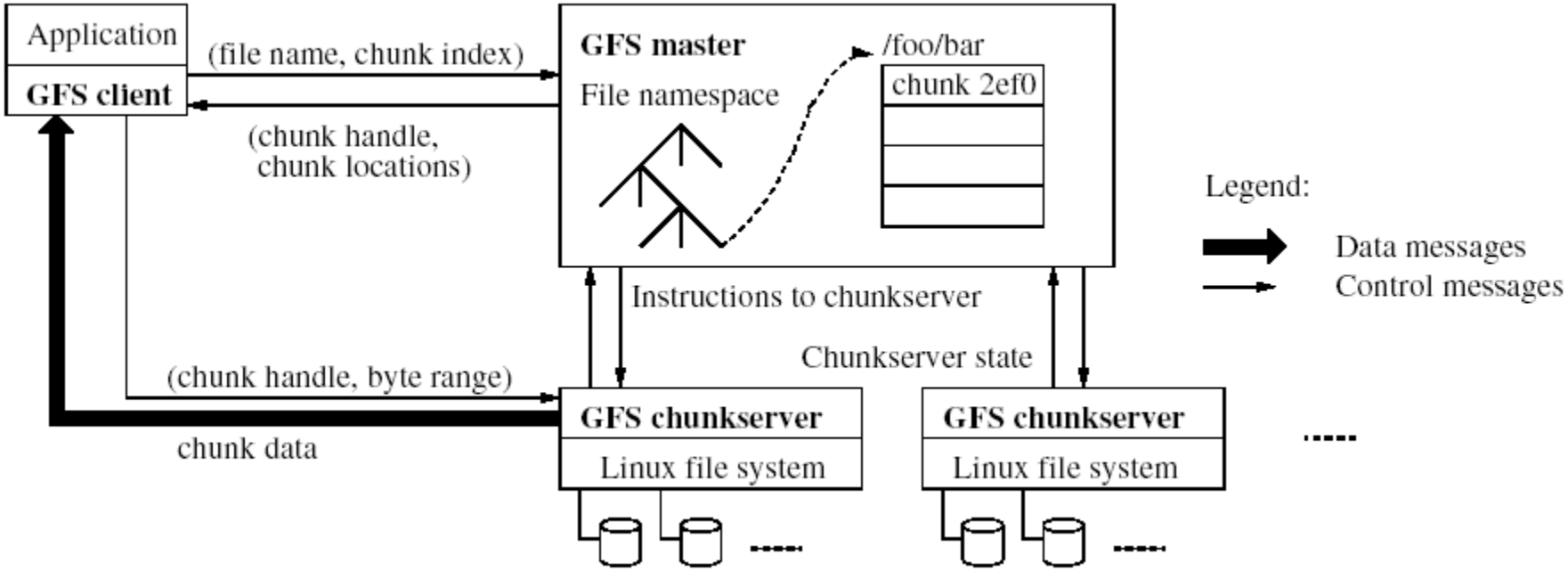
- **Idea for distribution and communication:**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Idea for coordination**
  - Only allow operations that minimize contention ("embarrassingly parallel")
  - Perform coarse-grained coordination
- **Map-reduce** addresses 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?
  - How to read a massive amount of data in a short time?
- **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

GFS Architecture

- Single master, multiple chunkservers



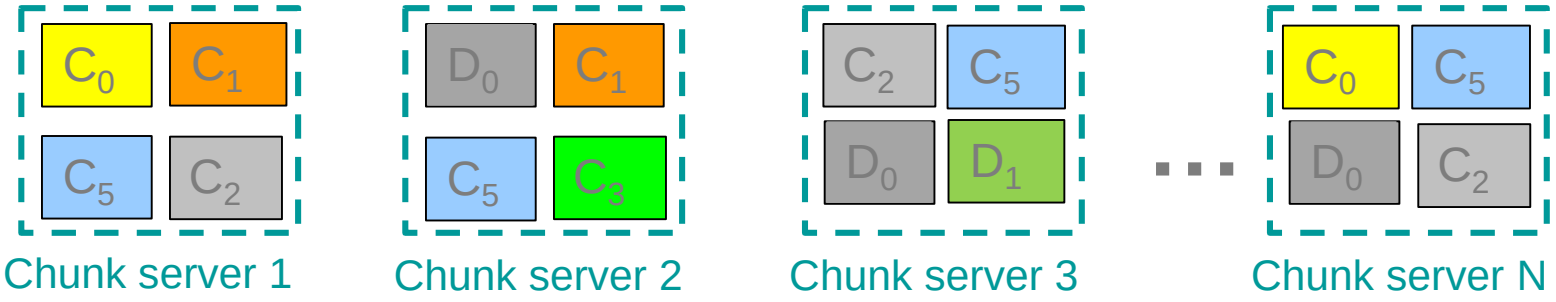
- To overcome single-point of failure & scalability bottleneck:
  - Use shadow masters
  - Minimize master involvement (large chunks; use only for metadata)

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 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



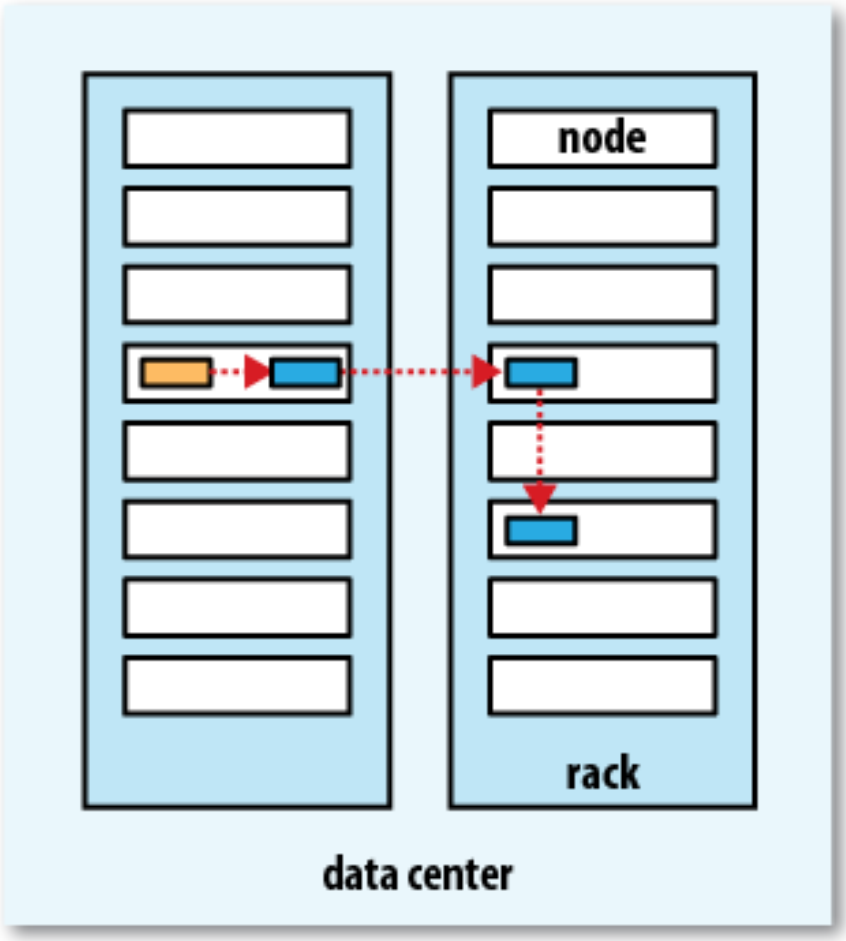
Bring computation directly to the data!

Chunk servers also serve as compute servers

Replica Placement

- Issues to consider:
  - reliability
  - write bandwidth
  - read bandwidth
  - block distribution.
- Hadoop's default strategy:
  - First replica: on the client node (or randomly chosen if client is outside the cluster)
  - Second replica: random, off-rack.
  - Third replica: same rack as second, different node.
  - More replicas: randomly chosen.

Example Replica Pipeline:



HDFS and Data Lakes

- HDFS use case: streaming data access
  - write-once, read-many-times pattern
  - time to read the whole dataset is more important
- HDFS is not a good fit for
  - low-latency data access
  - lots of small files
  - multiple writers, arbitrary file modifications
- HDFS is the shared common storage layer for multitude of computation frameworks
- "Data Lake": Collect raw, unstructured data (vs. Data Warehouse)
  - Consolidated Storage
  - Massive scalability
  - Original format of data
  - Unstructured Data, Semi-Structured-Data, Structured Data
  - Structured on demand or structure per application
- Competing Implementation technique: Key-Value / Object Stores

Programming Model: MapReduce

Warm-up task:

- We have a huge text document or huge collection of text documents
- Count the number of times each distinct word appears in the file(s)
- **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

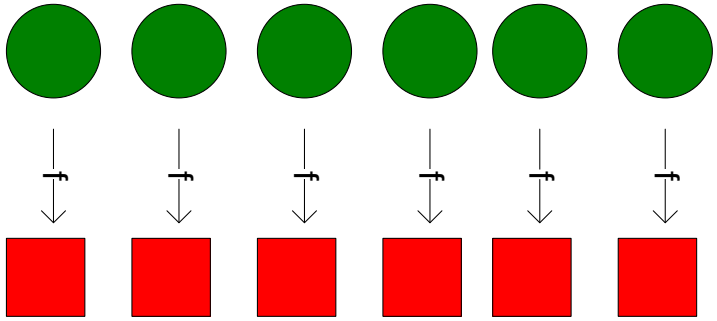
MapReduce: Overview

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

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

Background: map() in Haskell

- Create a new list by applying f to each element of the input list.



- **Definition of map:**

```
map :: (a → b) → [a] → [b]   -- type of map
map f []     = []            -- the empty list case
map f (x:xs)       = f x : map f xs      -- the non-empty list case
```

- **Example: Double all numbers in a list.**

```
Haskell-prompt > map ((*) 2) [1, 2, 3]
[2, 4, 6]
```

Implicit Parallelism in map()

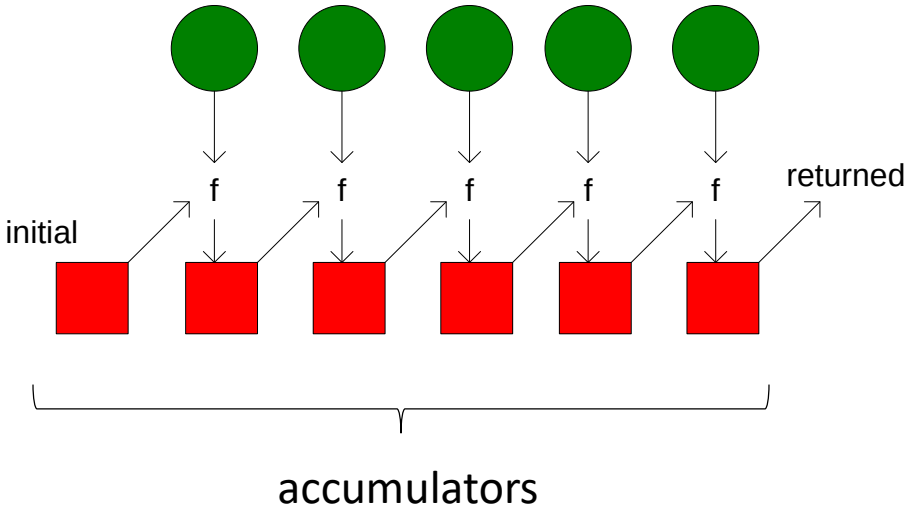
- In a purely functional setting, an element of a list being computed by map cannot see the effects of the computations on other elements.

- If the order of application of a function f to elements in a list is commutative, then we can reorder or parallelize execution.

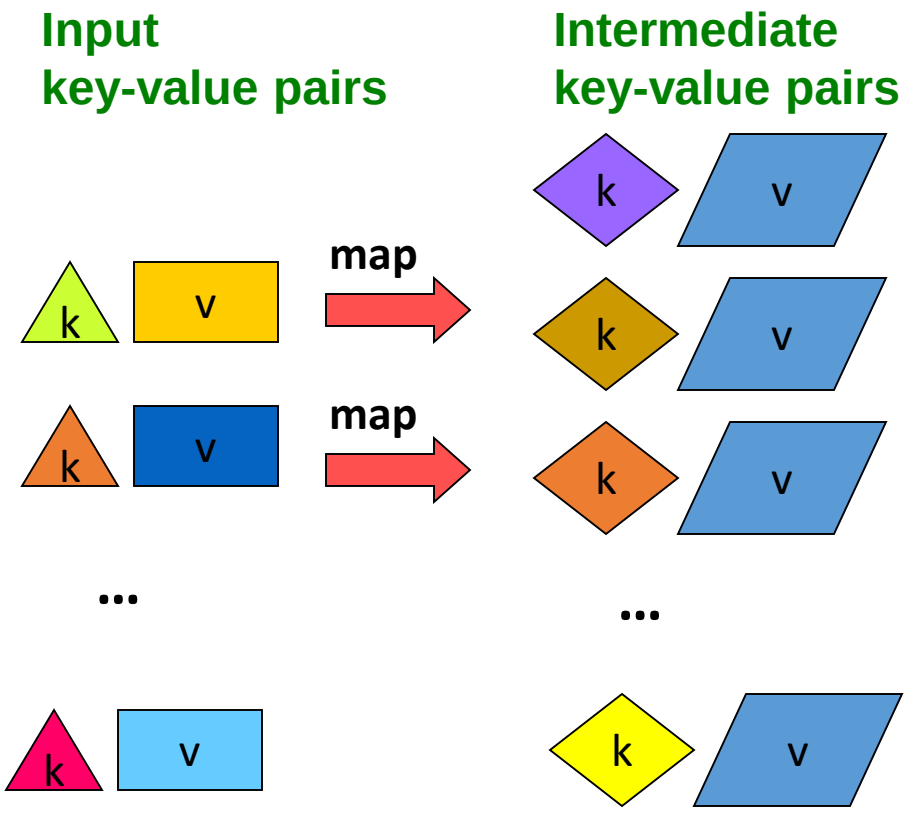
- This is the “secret” that MapReduce exploits.

fold()/reduce() in Haskell

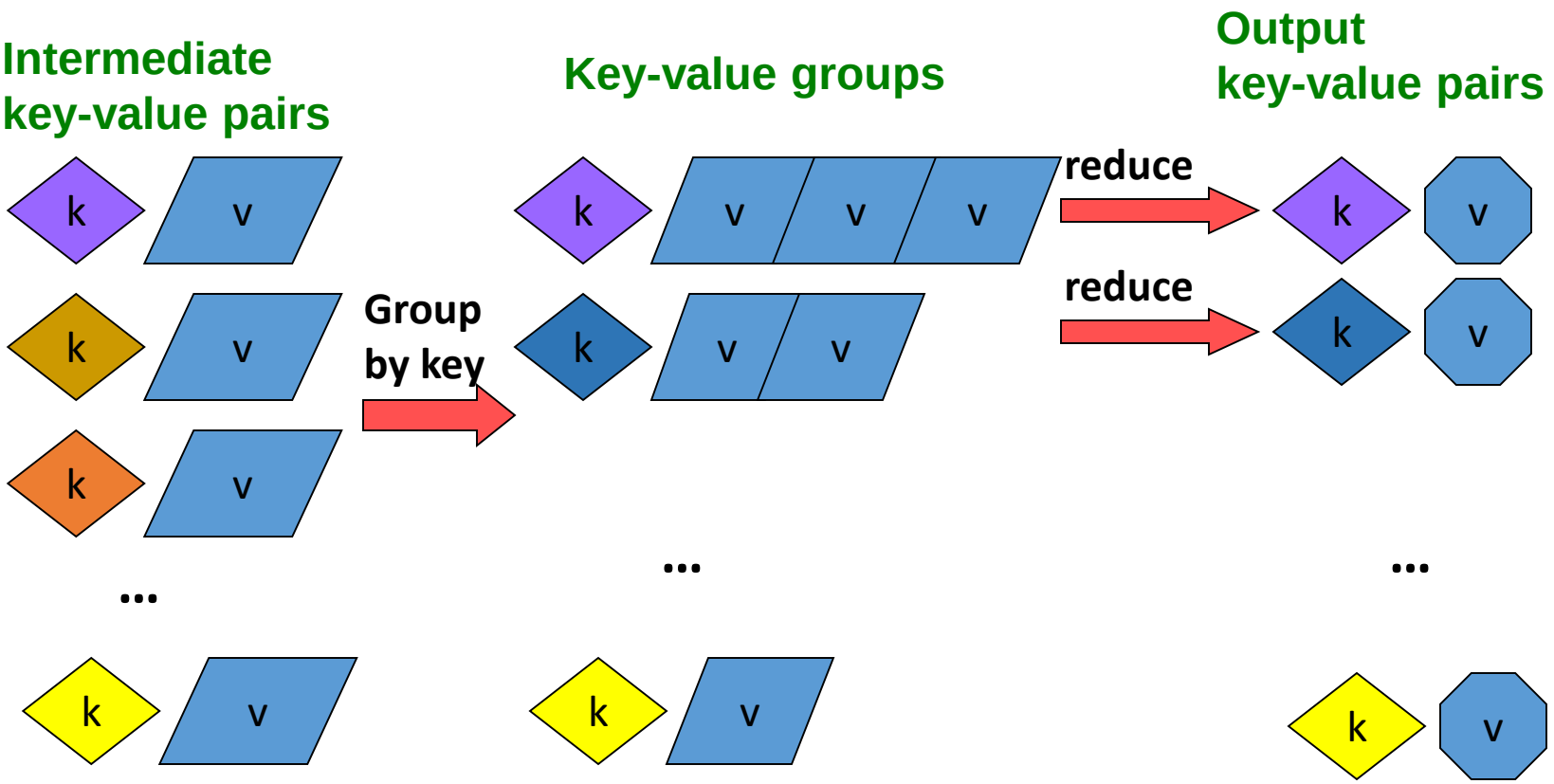
- Move across a list,
- Apply a function **f** to each element plus an **accumulator**.
- f returns the next accumulator value, which is combined with the next element of the list.



MapReduce: The Map Step



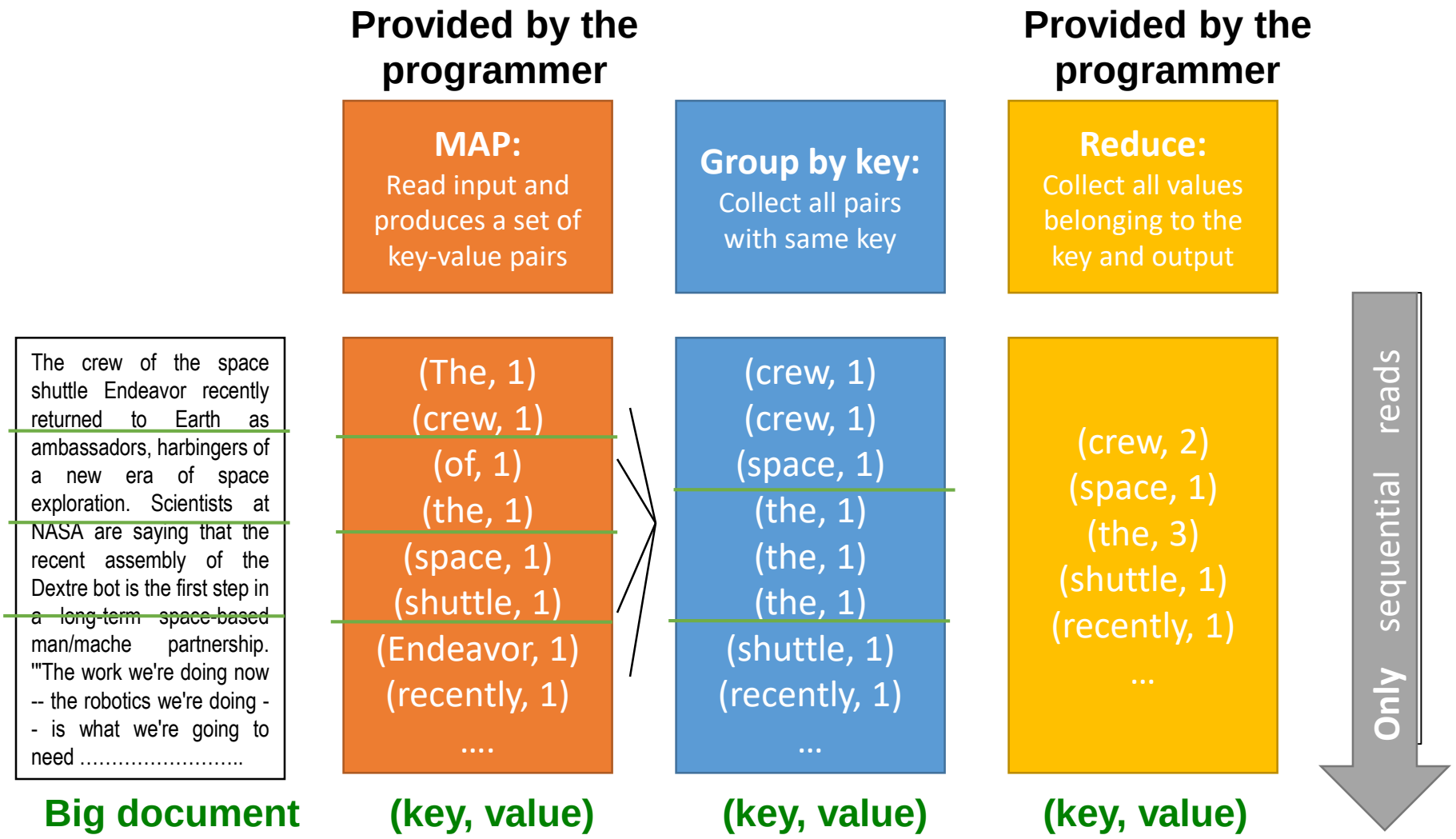
MapReduce: The Reduce Step



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: Environment

Map-Reduce environment takes care of:

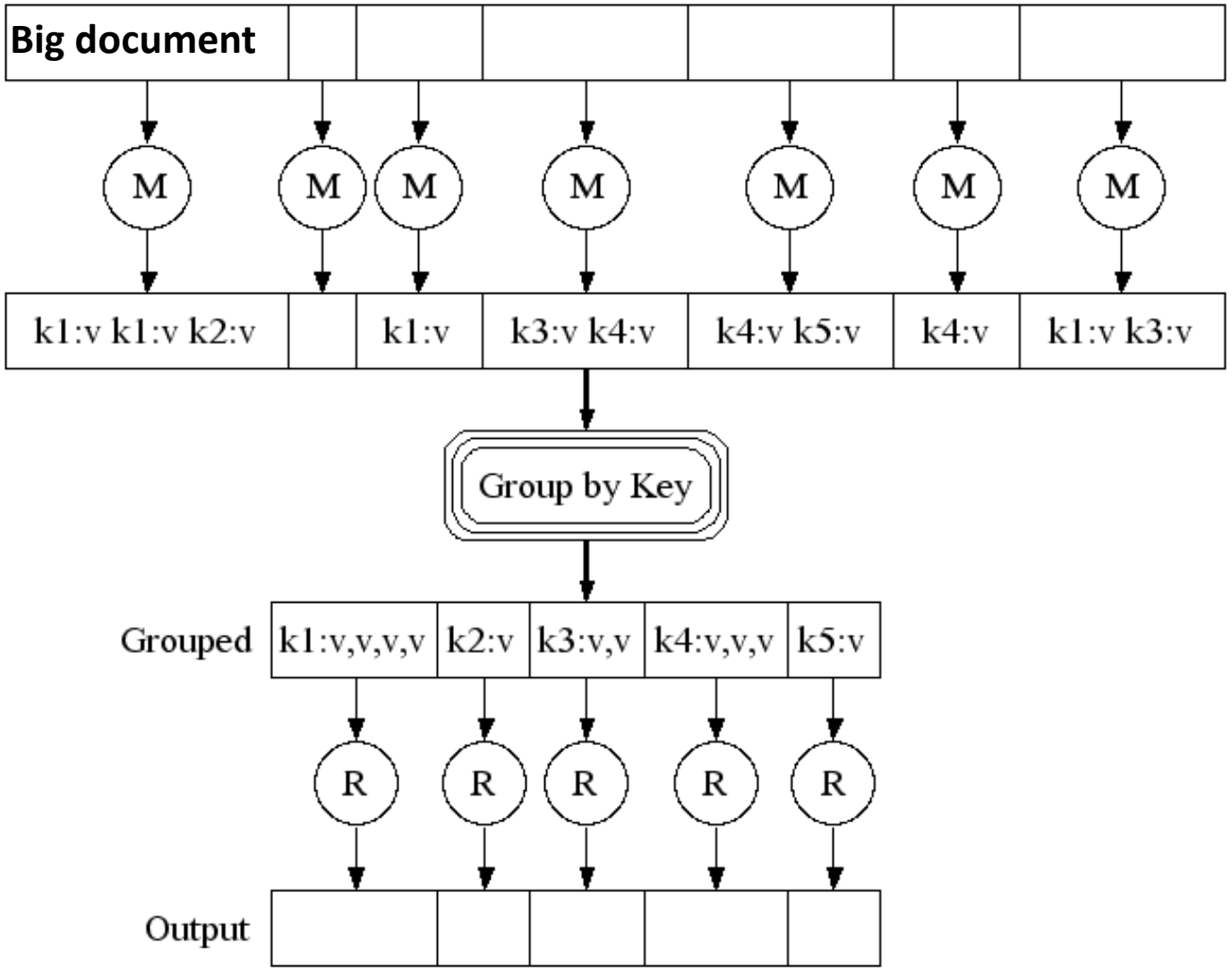
- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine failures
- Managing required inter-machine communication

Map-Reduce: Implementation overview

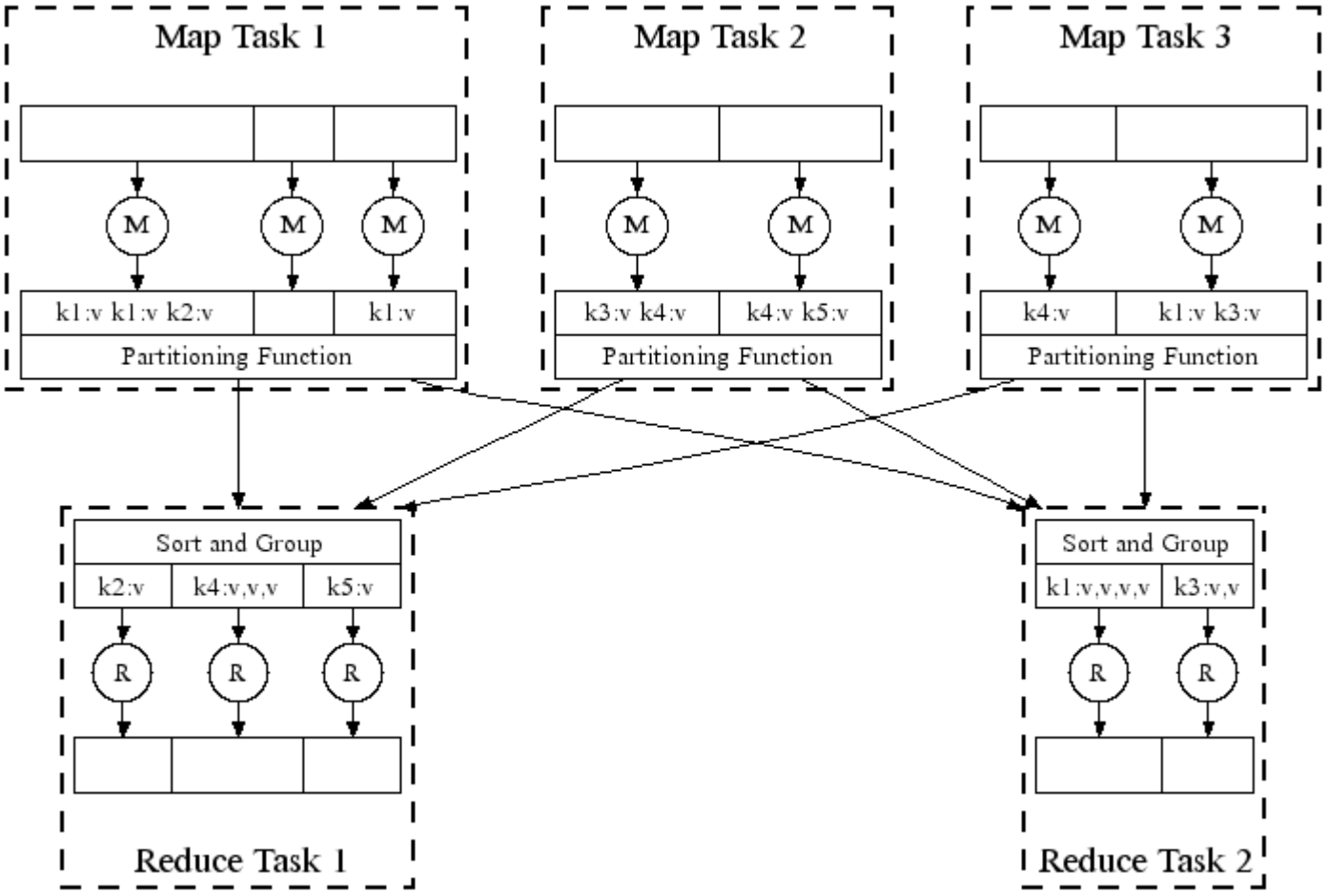
MAP:
Read input and produces a set of key-value pairs

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

Reduce:
Collect all values belonging to the key and output



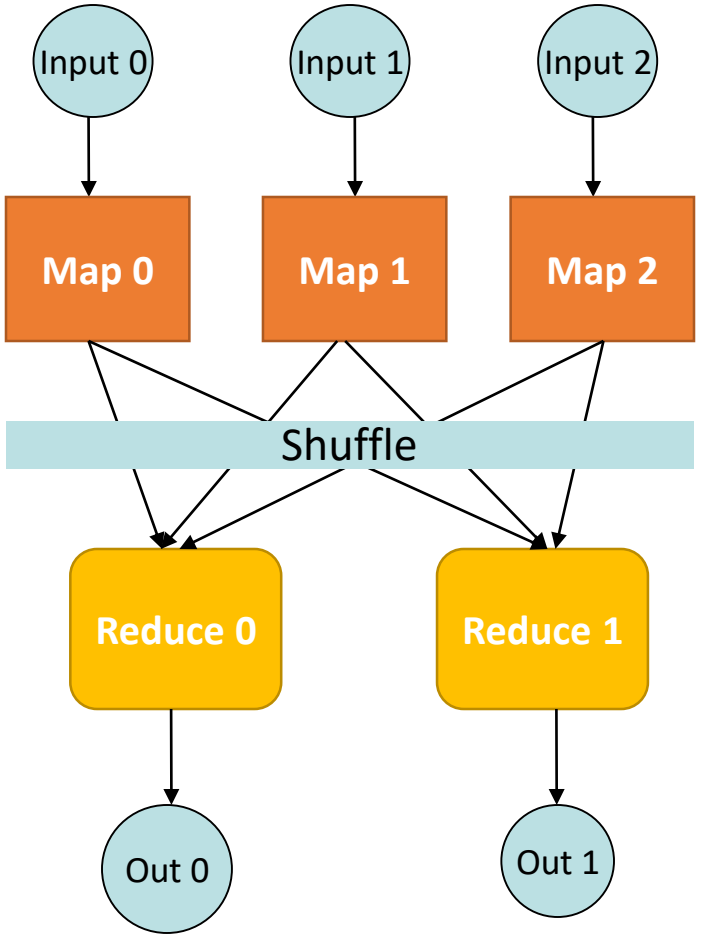
Map-Reduce: In Parallel



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 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
- Shuffle is a barrier/execution blocker



Data Flow

- **Input and final output are stored on a distributed file system (FS):**
  - Remember: data files are divided into 64 MB blocks and 3 copies of each are stored on different machines/racks
  - Put map() tasks physically on the same machine as one of the input replicas (or, at least on the same rack / network switch).
  - This way, thousands of machines can read input at local disk speed. Otherwise, rack switches would limit read rate.

- **Intermediate results are stored on local FS**
  - Trade off recovery time vs replication overhead
- **Output is often input to another MapReduce task**
  - MapReduce defines a plan of fixed length (no pipeline, no iteration)
  - Composition/Orchestration done from outside
  - Next MapReduce task needs to read from disk (materialization)
  - Optimization: partition/allocate output as needed by next 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

Dealing with Failures

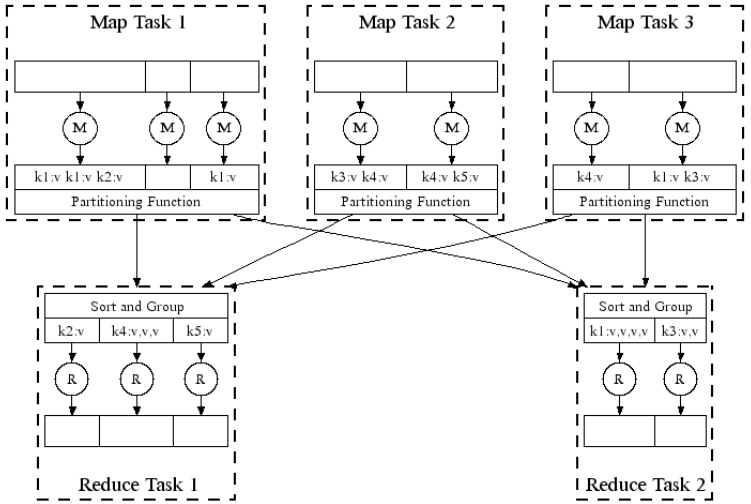
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce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

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

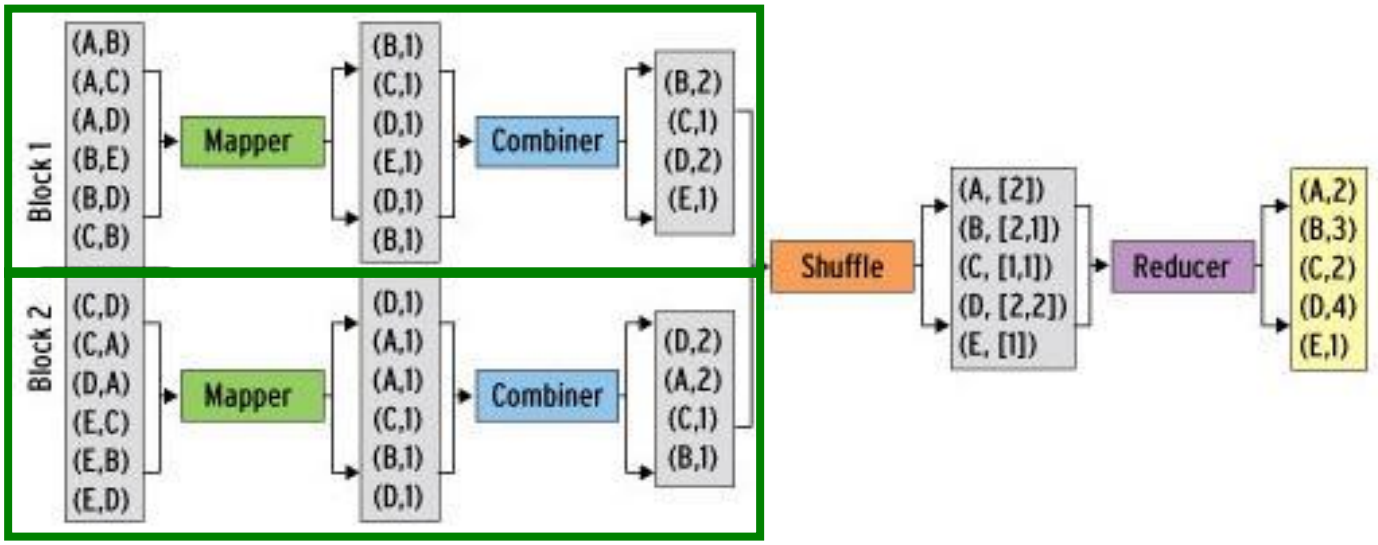
Refinement: Combiners

- 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):



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

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
  - ➢Large-Scale Matrix/Vector or Matrix/Matrix multiplications
- Caveat:
  - Not well suited for multi-stage or iterative problems

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 (Reduce-Side 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)*.

- Group By can be expressed in the same way
- Set operations also fit onto a similar pattern (duplicates, intersection, …)
- Selection, Projection fit onto Map

Cost Analysis Approaches

Cost Measures for Algorithms

Classical analysis: Big-O notation

- Metric: computation steps (e.g., comparisons, additions)
- Asymptotic/limiting behavior to input size
- Fastest-growing component

Not the only / most useful measure in
big-data, elastic workloads
(adding more machines is always an option)

In MapReduce we quantify the cost of an algorithm using

1. *Communication cost* = total I/O of all processes (disk + network)
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost**
    - input file sizes
    - 2 × (sum of the sizes of all files passed from Map processes to Reduce processes)
    - sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of
    - the largest input any map process
    - the largest 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

- Total cost tells what you pay in rent from your friendly neighborhood cloud

- Elapsed cost is wall-clock time using parallelism

Cost of a Map-Reduce Join

- **Total communication cost**
  = O(|R|+|S|+|R ⋈ 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 comm. cost

Wrap Up

- Evaluating massive data sets often leads to distributed storage and computation
- Storage – Distributed File Systems (GFS/HDFS):
  - Large files broken into fixed-size chunks (64-256 MB)
  - Replicate for redundancy, network locality
  - Write-once (often append), read-many
  - Data Lakes: Store "raw" and partially processed data
- Computation – MapReduce
  - Computation close to data
  - Framework based functional programming concepts
  - Restrictive data access, limited execution model
  - Hide all systems aspects (distribution, failure,…)