MapReduce
Large Scale Data Mining

Helge Holzmann, Avishek Anand

L3S Research Center, Hannover, Germany

14/04/2016

based on Mining Massive Datasets

by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University)

http://www.mmds.org

Code examples on https://github.com/helgeho/MapReduceLecture

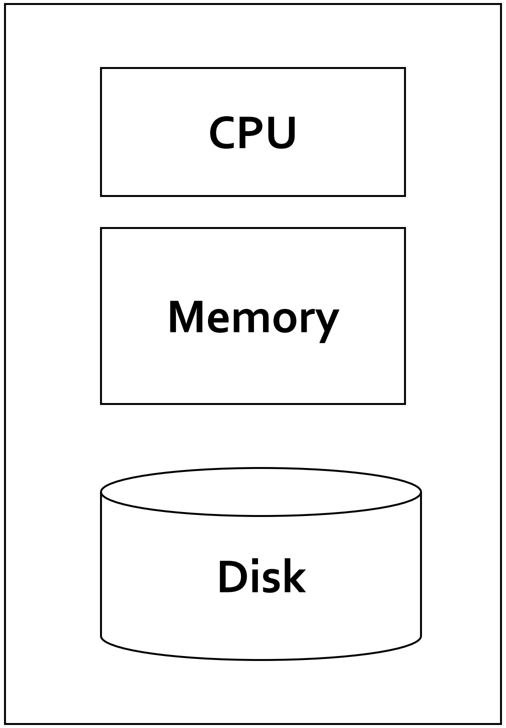
What are we going to talk about?

- **Programming**
  from imperative, procedural; to functional; to parallel; to distributed; to MapReduce
- **Distributed computations and file systems**
- **Problems and algorithms**
- **Refinements, extensions, alternatives**
- **… many buzzwords**
  Hadoop, Pig, Hive, Spark, … is it Pokemon or BigData?
  https://pixelastic.github.io/pokemonorbigdata

MapReduce

- Much of the course will be devoted to **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

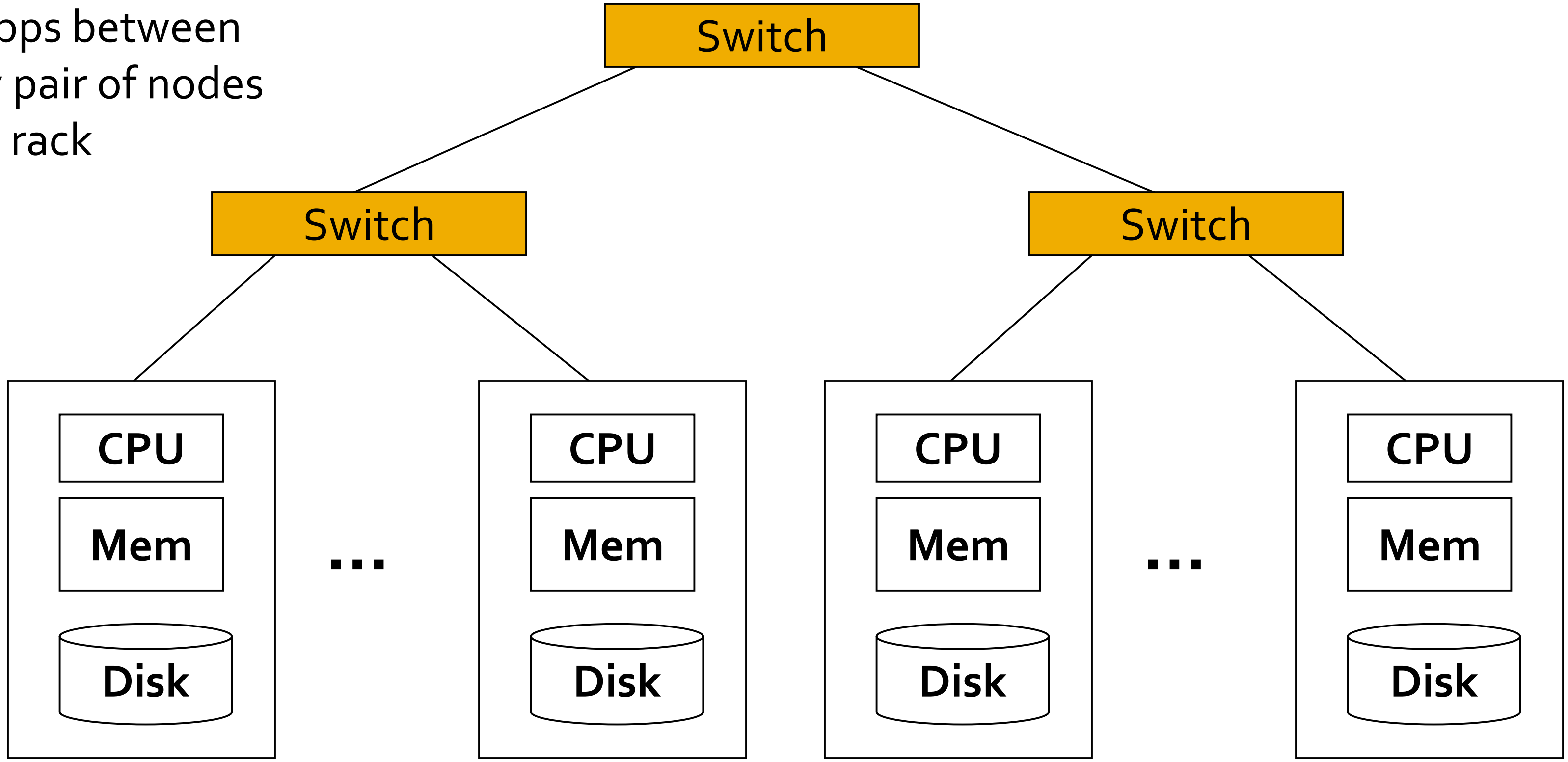
Motivation: Google Example

- 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

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!

Example: Sum of Number Strings

- **SUM("two", "seven", "one", "five") = 15**
  1. Map strings to numbers: two → 2, seven → 7, one → 1, five → 5
  2. Sum (reduce): 2 + 7 + 1 + 5 = 15
- **Assumption:** Mapping is expensive
- **Implementations (from imperative to functional):**
  - Sequential: *Sequential.java*
  - Multi-threaded, concurrent: *Threaded.java*
  - Multi-threaded, synchronized: *Synchronized.java*
  - Multi-threaded, parallel: *Parallel.java*
  - Multi-threaded, parallel, refactored: *ParallelRefactored.java*
  - Functional, parallel (in Scala): *NumberSum.scala*

Implementations: Sum of Number Strings

- **Imperative / Sequential**
  - Not easily parallelizable
- **Concurrent**
  - **Challenge: side effects**
- **Synchronized**
  - Losing parallelism
- **Functional / Parallel**
  - No side effects
  - Easily parallizable

Think functional!

Functional Implementation in Scheme / Lisp

- No side effects, less verbose

```
(define (number sym)
    (define (get-index sym index list)
        (if (eq? (car list) sym)
            index
            (if (pair? (cdr list))
                (get-index sym (+ index 1) (cdr list))
                -1)))
    (get-index sym 0 (list 'zero 'one 'two 'three 'four 'five 'six 'seven 'eight 'nine)))

(fold-left
    + 0
    (map
        number
        (list 'two 'seven 'one 'five)))
```

(reduce is performed by fold-left here)

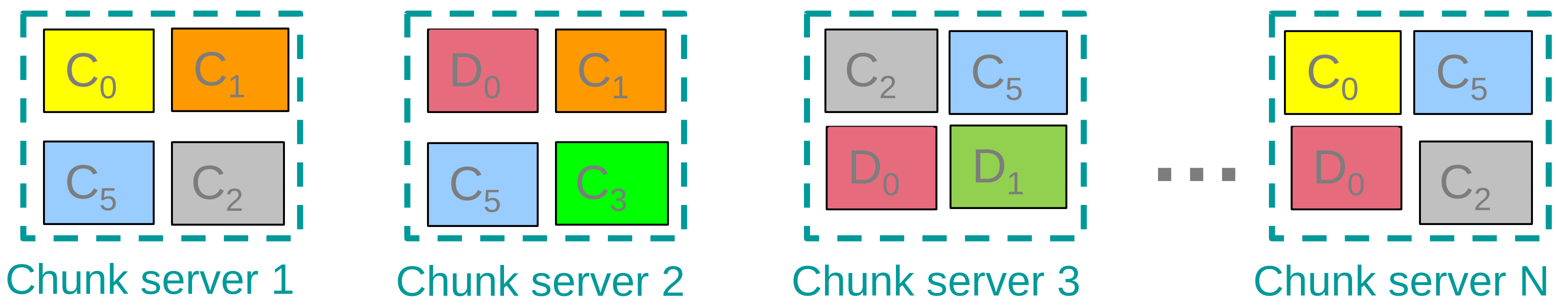
Think functional!

From Parallel to Distributed

- **From one to multiple machines**
- **Distributed the data**
  - Distributed File System
- **Distribute the computation**
  - Loosely coupled code, distributed computing, central coordination
- **Previous example:** Sum of number strings
  - Hadoop Implementation: *NumberSumHadoop.java*

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

Distributed File System

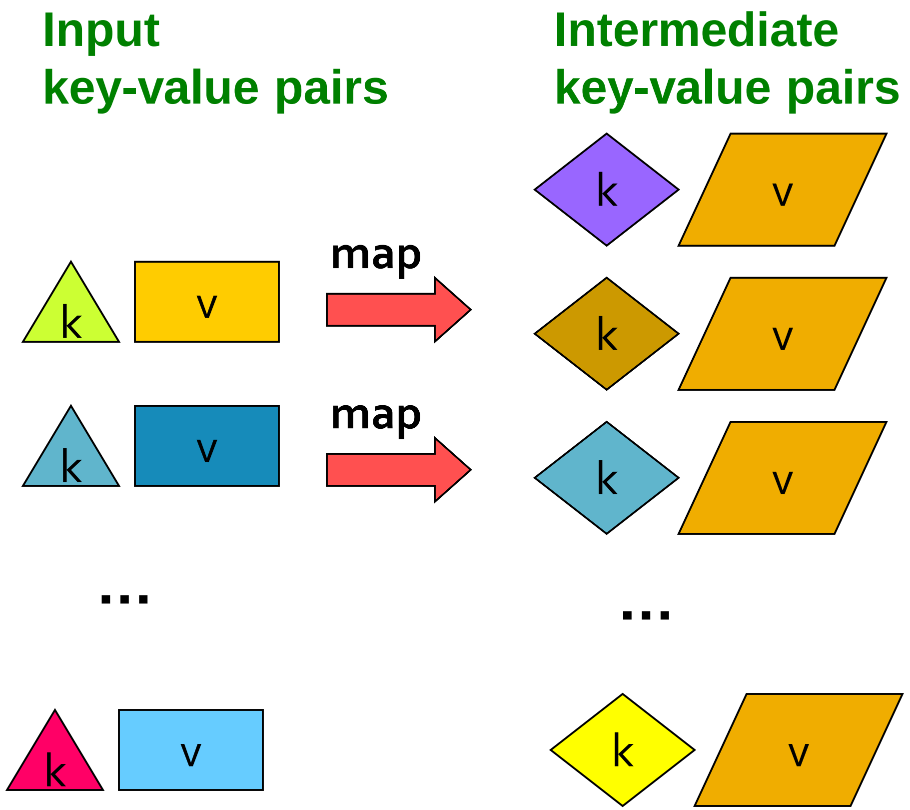
- Chunk servers
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master node
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might be replicated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

MapReduce: Overview

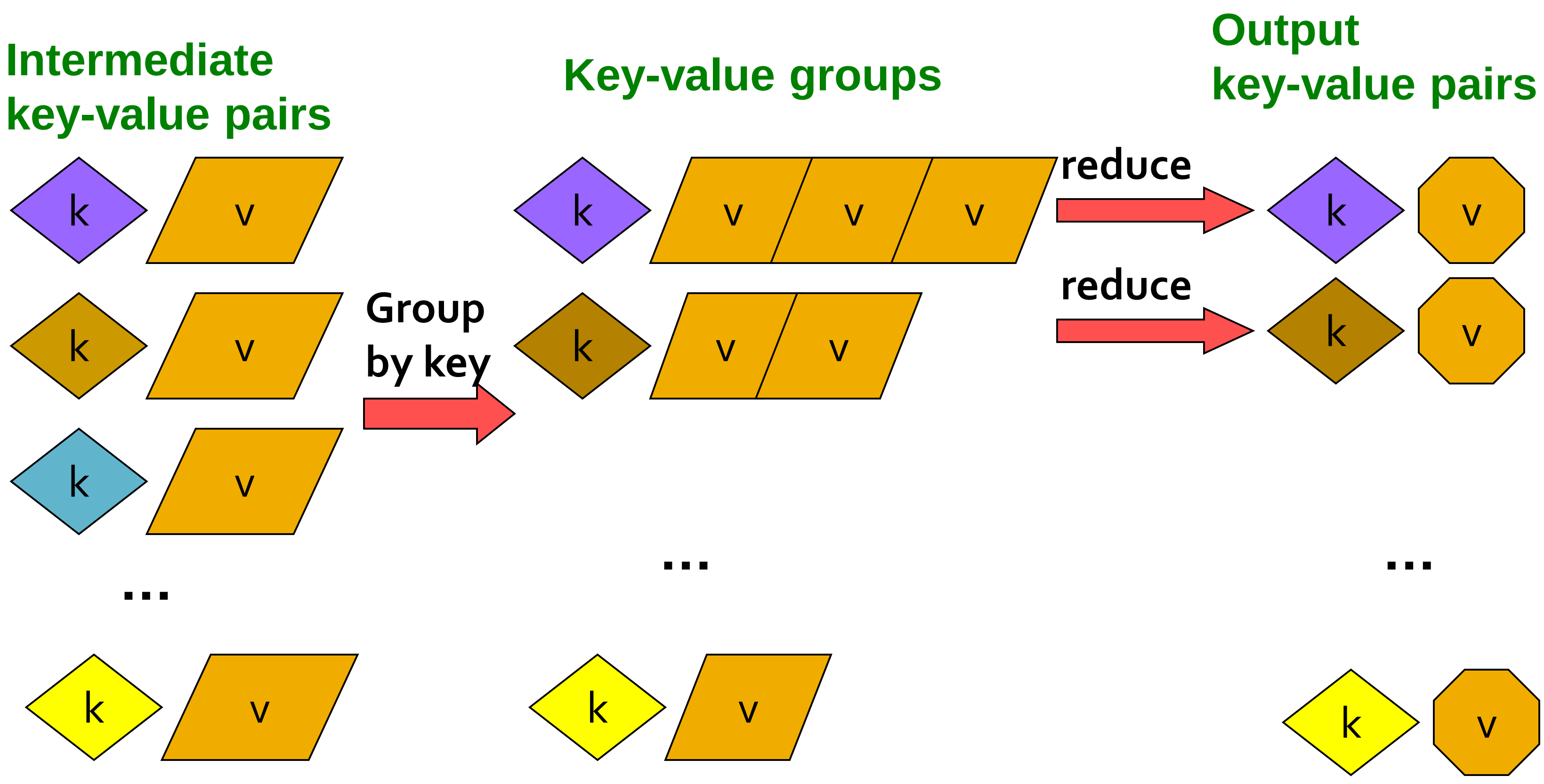
- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

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

MapReduce: The Map Step



MapReduce: The Reduce Step

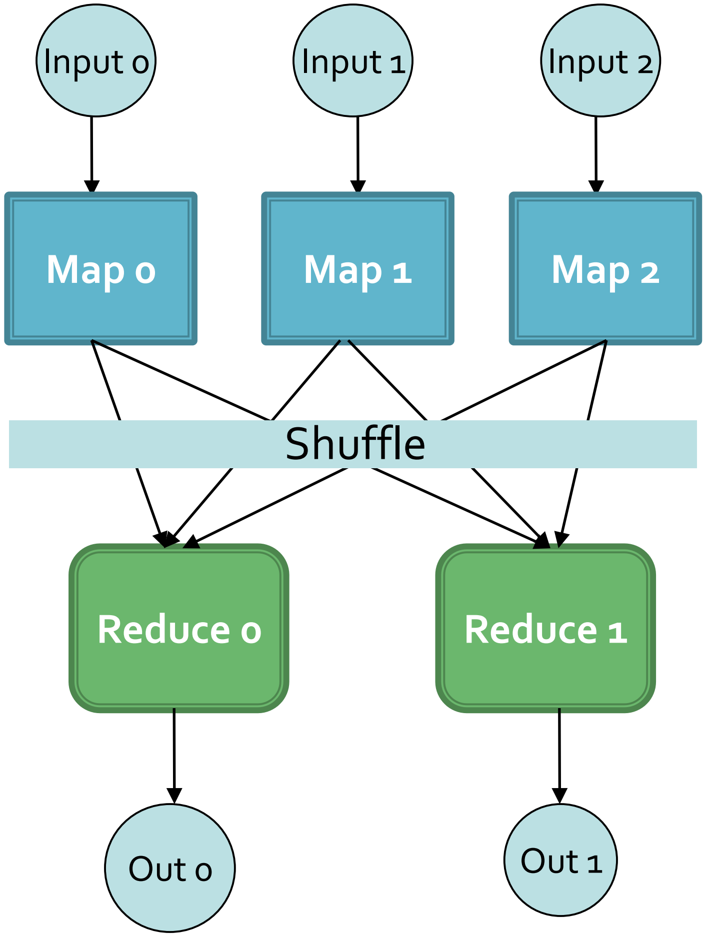


More Specifically

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

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

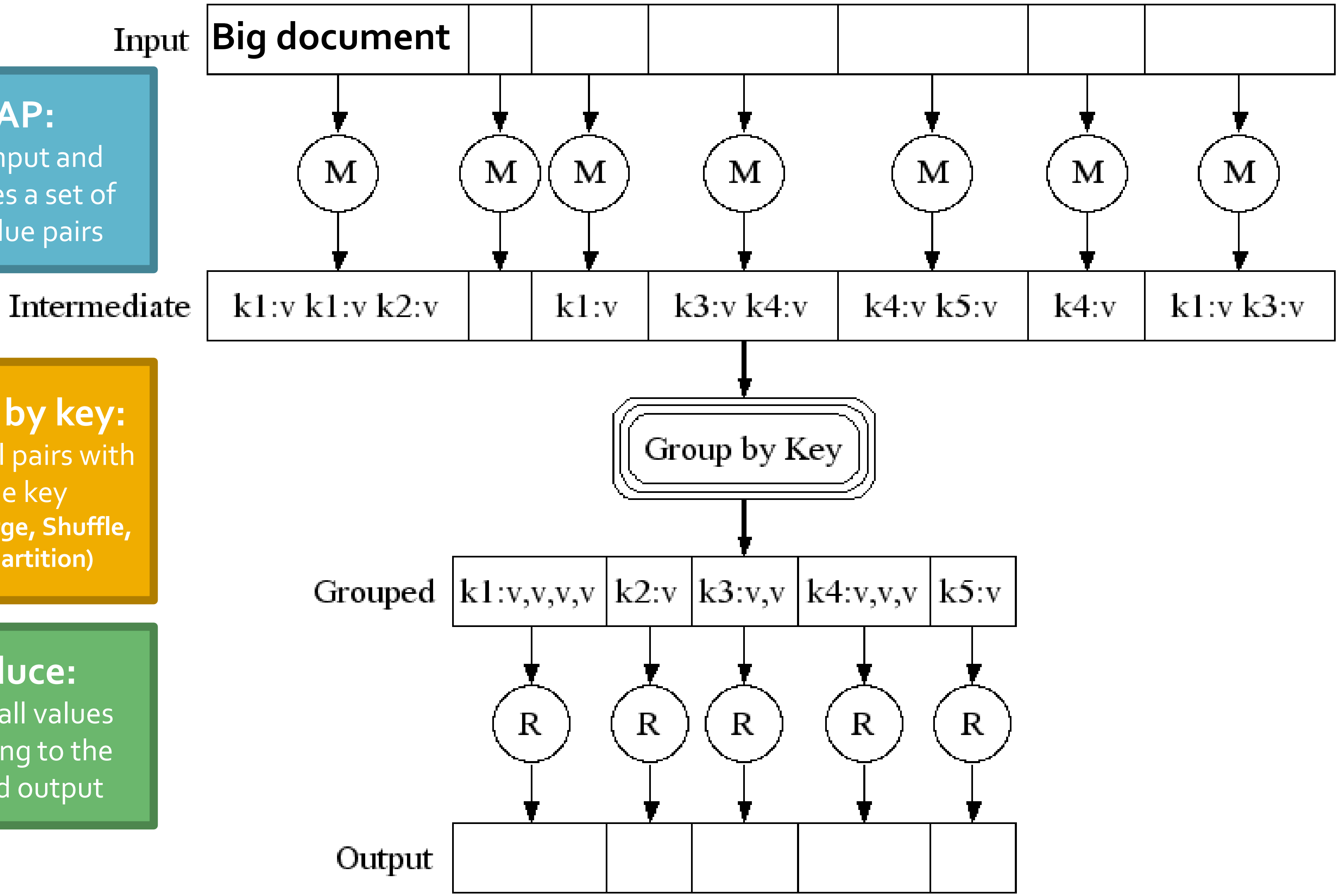


Map-Reduce: A diagram

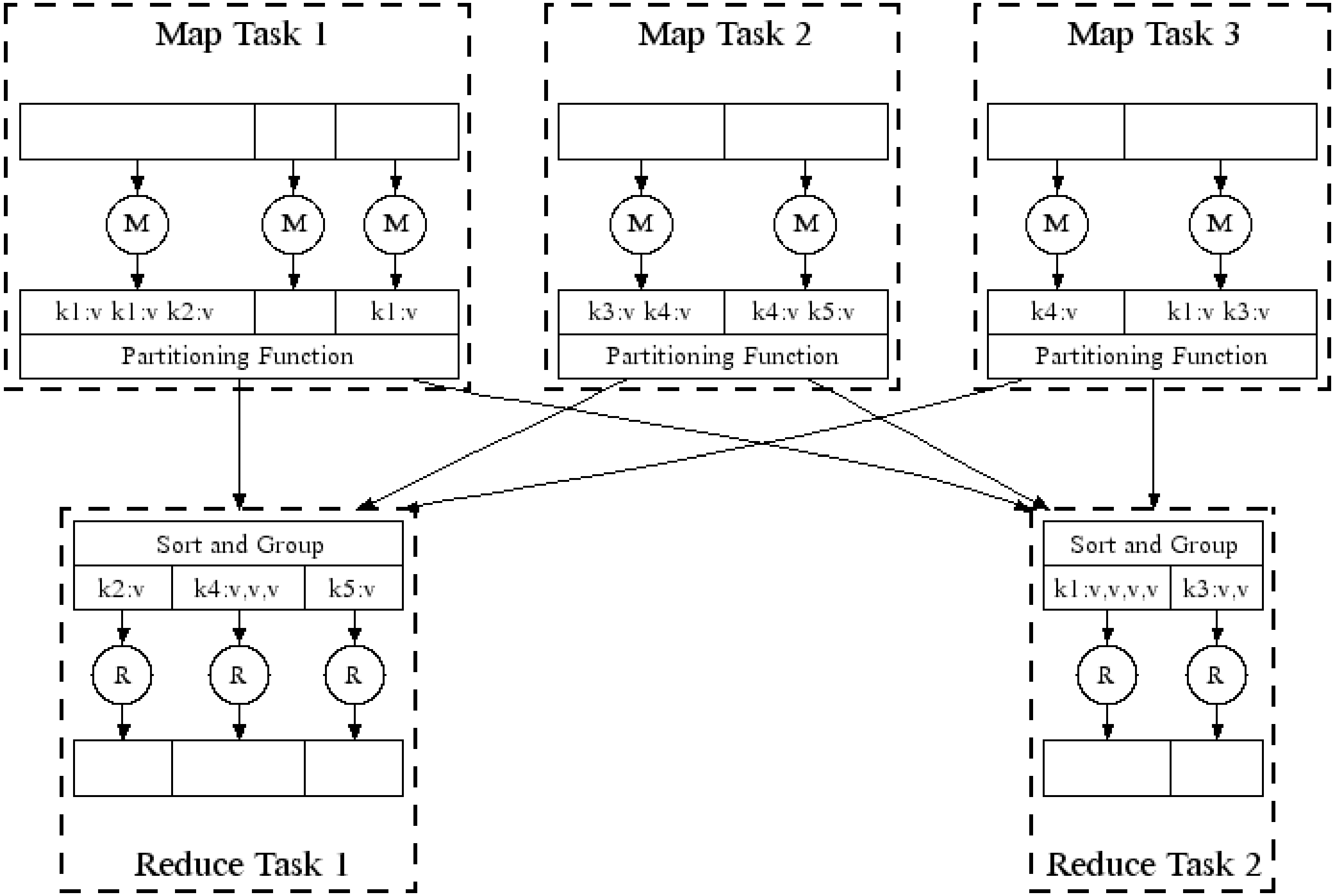
MAP:
Read input and produces a set of key-value pairs

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: 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

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**

Implementations

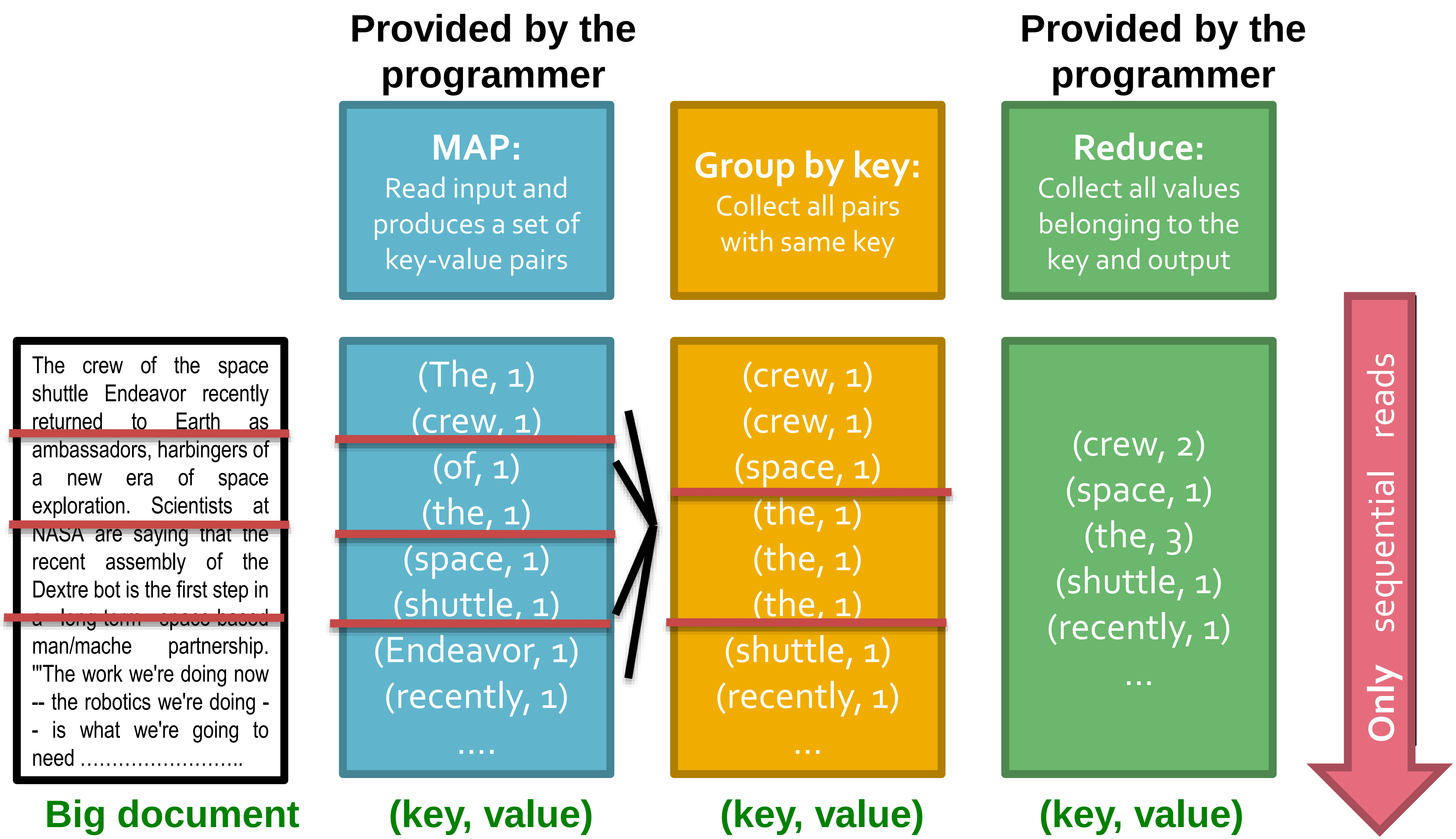
- Google
  - Not available outside Google
- **Hadoop**
  - An open-source implementation in Java
  - Uses HDFS for stable storage
  - Download: http://lucene.apache.org/hadoop/
- Aster Data
  - Cluster-optimized SQL Database that also implements MapReduce

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

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)
```

Example: Word Count

- **Input:** (big) text file(s), one sentence per line, split per line
- **Mapper input:** one sentence
- **Mapper output:** (word, 1) pairs
- **Reducer input:** one word, corresponding pairs
- **Reducer output:** (word, count) pairs
- **Implementations:**
  - Functional: *WordCount.scala*
  - Hadoop: *WordCountHadoop.java*

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

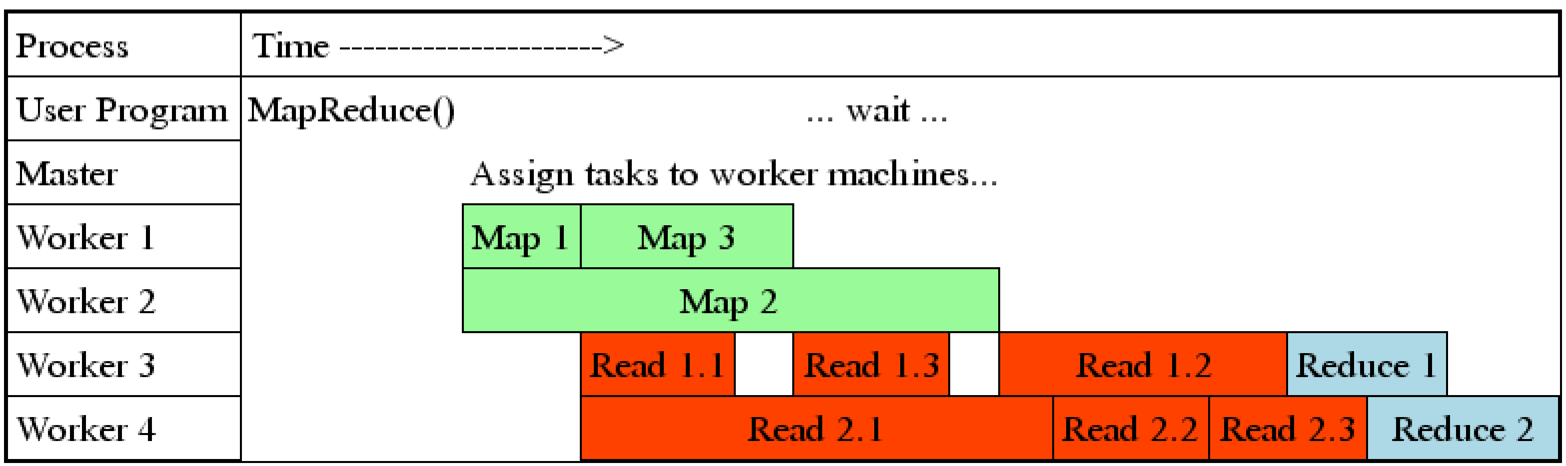
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce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

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

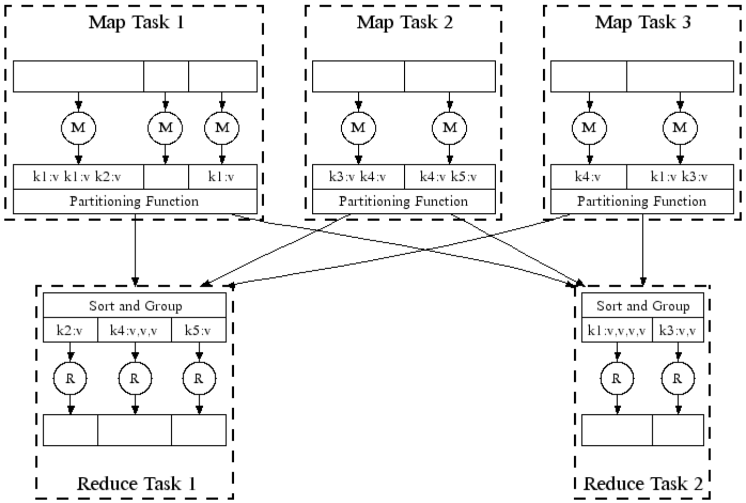


Refinements: Backup Tasks

- **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

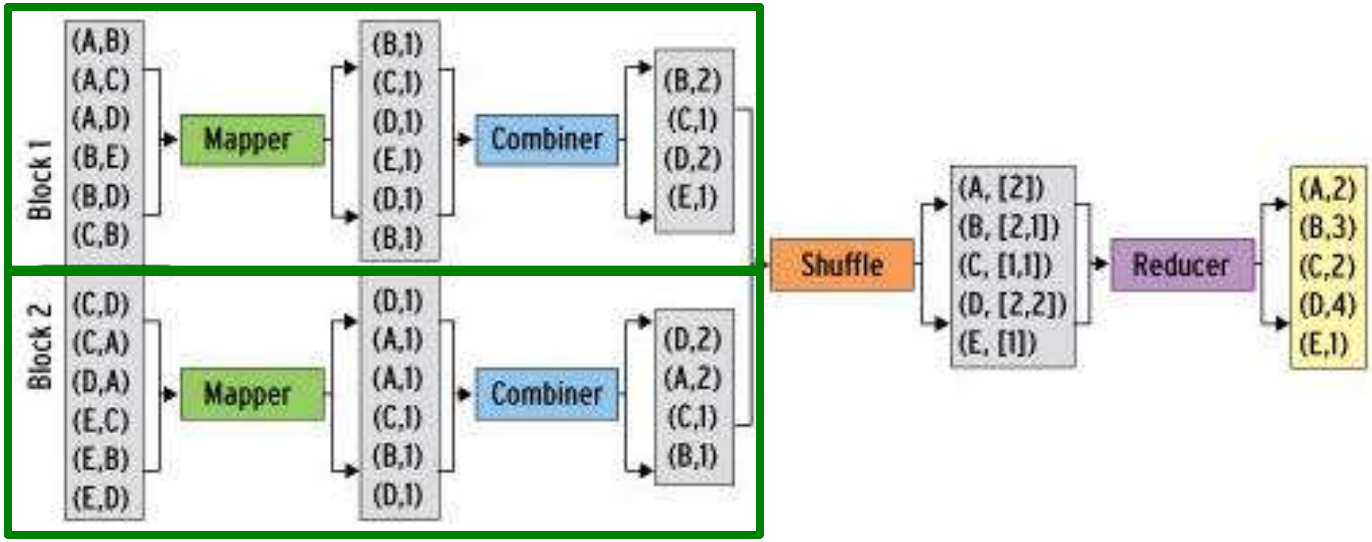
Refinement: Combiners

- 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: Sorting with MapReduce

- **Sorting reduce values between Map and Reduce:**
  - Mapper → Shuffle (Partition, Sort, Group) → Reducer
- **Map output:** (key#secondary_key, value)
  - E.g., sort "col1 col2 col3 col4" on third column: ("key#c", "a b c d") "key#col3" is called a composite key, with 'key' called natural key
- **Partition:** Ensure same natural key goes to same reducer
- **Sort:** Sort records at the reducer by secondary key
- **Group:** Group records by natural key
- **All steps can be customized to implement any sorting**
- **Global sorting:** one reducer or sorted partitioning
  - One reducer only for small data, sorting partitions not trivial

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)*.

Chaining MapReduce Jobs

- **Required for chaining multiple operations:** such as Join, Group, Aggregate, Sort, …
- **Simplified by tools, such as *Hive, Pig, Spark*, …**
- ***Hive* allows SQL (HiveQL) queries**
  - E.g., `SELECT SUM(input1.field1) AS sum, COUNT(*) AS count FROM input1 JOIN input2 ON (input1.field2 = input2.field1) GROUP BY input2.field2 ORDER BY input2.field2 DESC;`
  - Gets translated into multiple MapReduce jobs
- **Example:** Inverted index creation
  - Implementation with *Pig*: *IndexCreation.pig*
  - Gets translated into multiple MapReduce jobs

Example: Matrix Multiplication

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \times \begin{bmatrix} n_{11} & n_{12} & N_{13} \\ n_{21} & n_{22} & n_{23} \\ n_{31} & n_{32} & n_{33} \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & P_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix}$$

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

- $p_{\mathbf{ik}} = m_{\mathbf{i}1}n_{1\mathbf{k}} + m_{\mathbf{i}2}n_{2\mathbf{k}} + m_{\mathbf{i}3}n_{3\mathbf{k}}$
- Chain two MapReduce jobs:
  - 1. Natural join, multiply values in reducer
    - Output values of reducer for key j: all pairs (*i#k*, $m_{ij}m_{ij}$)
  - 2. Group by key, sum values in reducer
    - Output value of reducer for key *i#k*: (*i#k*, $\sum_j m_{ij} n_{jk}$)

Cost Measures for Algorithms

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

1. *Communication cost* = total I/O of all processes
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count 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.
  - **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
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

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 comm. cost

Spark vs. Hadoop/MapReduce

- **Spark has recently become a very popular alternative**
  - Supports MapReduce computing model: map, reduce, …
  - Up to 100x faster than Hadoop (s. http://spark.apache.org)
  - Spark for Web archives @ L3S: *ArchiveSpark* https://github.com/helgeho/ArchiveSpark
- **Extensive use of main memory vs. disk**
  - Lower communication costs
  - Fault tolerance by lineage / recovering vs. replication
- **Functional interface and lazy transformations**
  - Transformations chained and defered until action (e.g., reduce, …)
- **Previous example:** Inverted index creation (cp., Pig)
  - Spark implementation (in Scala): *IndexCreation.scala*

Summary

- **MapReduce:** distributed computing model
  - Think functional!
- **Distributed file system replicates data**
  - Provides fault tolerance
- **Computation exploits data locality**
  - Computing where the data is stored
- **Refinements enable flexibility and optimizations**
  - Combiners reduce at the mapper, shuffling allows for sorting
- **Chaining MapReduce tasks for larger algorithms**
  - Tools can help, but it is crucial to understand the operations

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]