Questions about This Lecture + @

- 질의 응답은 어떻게?
  - 일단, 이메일 (djchoi@inu.ac.kr) 로 보내주시면, 모두 취합해서 가능한 강의 때 동영상으로 찍어 올릴 것임.
  - 이외의 개인적으로 물어보는 질문들은 개별답변 혹은 office hour를 이용해주시기를 바람.
  - (Q&A를 하는 시간을 따로 갖는 것이 좋은지는 아직 고민중)

- Basic Operation 강의 (~4주차) 관련,
  - 다소 의미없게 느껴질 수 있음.
  - "지식" 보다 그 지식이 발생하게 된 문제 상황, 해결 concept을 주로 이해부탁

Lecture 2: MapReduce

Instructor: Daejin Choi (djchoi@inu.ac.kr)

INCHEON NATIONAL UNIVERSITY

Contents

- MapReduce: An overview
- MapReduce Problems
- Design MapReduce Framework

MapReduce: An Overview

Large Scale Computing for Data Mining

- Process a lot of data to produce other data

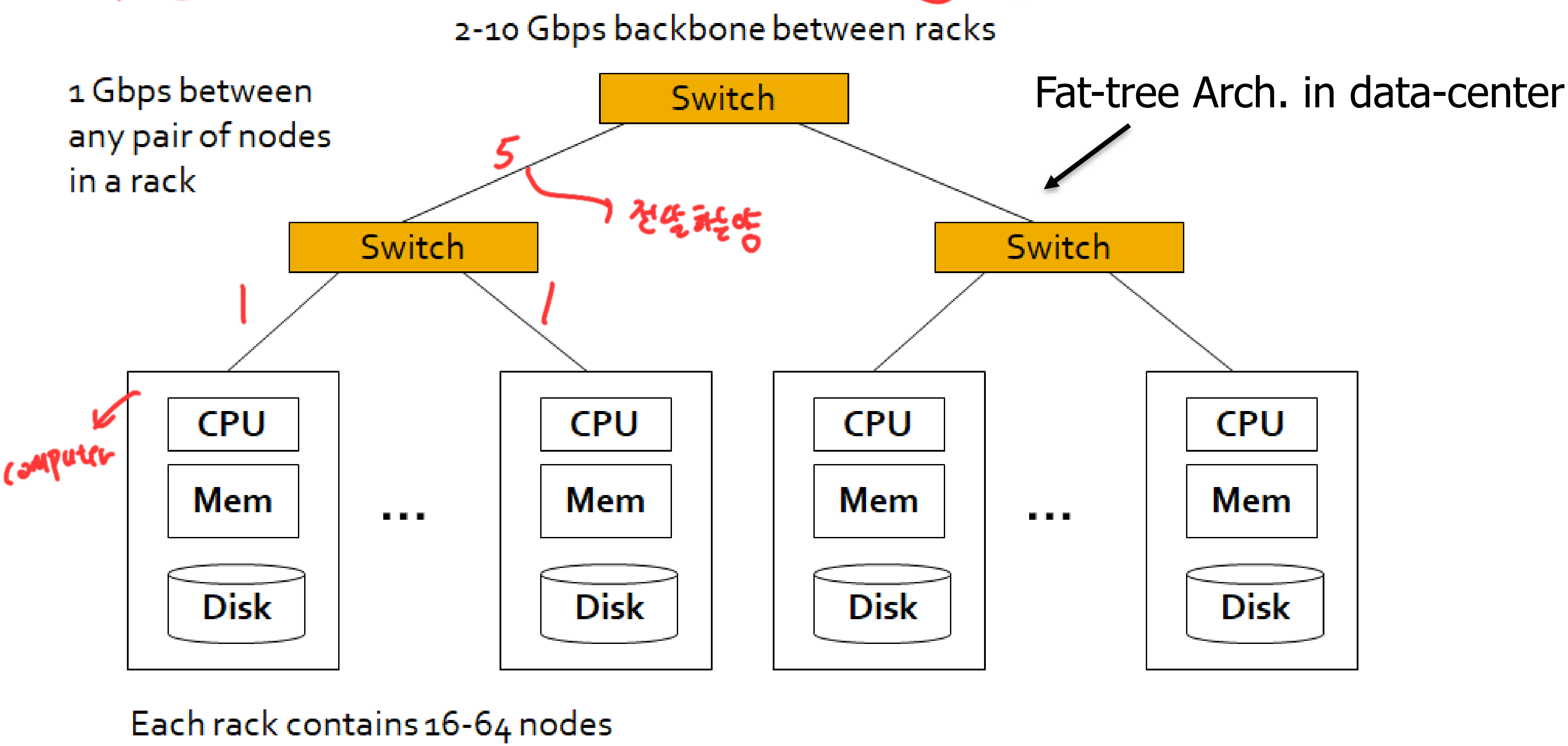
- Requirements?
  - CPU, Memory, HDD, …

  → Is single machine enough? NO!

- 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!

  → Using machine clusters is essential

- **Happily, standard architecture for such problems has been emerging:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them



Resources Are Enough, but …

- Programming (or operation) matters
  - Traditional programming is serial, so resources cannot be used fully

- Mining big data requires large-scale computing whose key component is **parallel programming**
  - Break processing into parts that can be executed concurrently on multiple processors (or machines)

- It is not as easy as you expect
  - NOT all problems can be parallelized

  → How do you design and distribute computation?
  (Another requirements: Network)
  - Node failure

Basic Operation: MapReduce

- Created by Google (Jeffrey Dean and Sanjay Chemawat) in 2004

- Inspired from LISP
  - **Map** (function, set of values) ⇒ python map이랑 같음
    - Applies the given function to each element of the set

    (map 'length '(() (a) (a b) (a b c)) => (0 1 2 3)

    병렬 가능
  - **Reduce** (function, set of values) ↳ 연산

    (reduce #'+ '(1 2 3 4 5)) => 15

    병렬 불가

- It becomes common!
  - Python, R, …

MapReduce is also Framework!

- Apache Hadoop MapReduce, Amazon Elastic MapReduce
- Framework for parallel computing, dealing with a lot of issues like
  - Parallelization
  - Data distribution
  - Load balancing (operation, data)
  - Fault tolerance
    오류안정성
- Programmers get simple API
- Allows one to process huge amounts of data (terabytes and petabytes) on thousands of processors

MapReduce as a Programming Model

- Sample problem
  - Find top 10 URLs by access frequency by analyzing log files from web server(s)

- The problem can be converted as **WordCount**
  - Counting words in a huge text document

WordCount

- Assume that a file is too large for memory, but all <word, count> pairs fit in memory

  → So, we have to divide the tasks into multiple sub-tasks

1) load a part of the file, 2) count word	**MAP**
3) then, merge	**REDUCE**

- Each sub-task can be run in parallel

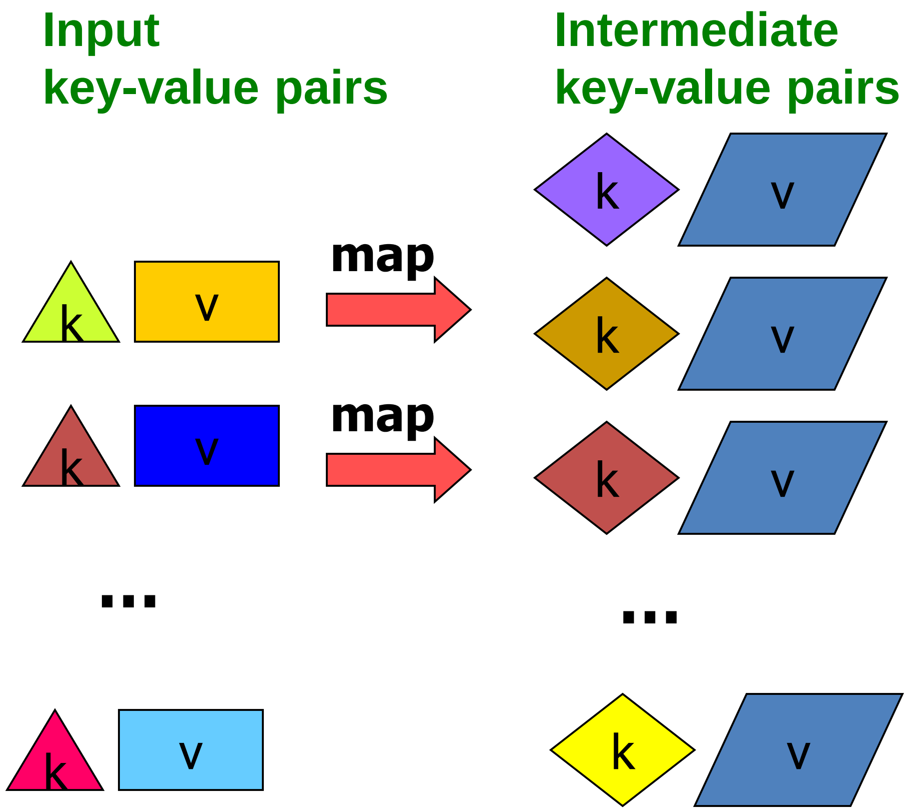
  → Reduce not only the resource usage (per minute), but execution time!

Let’s Generalize Formally

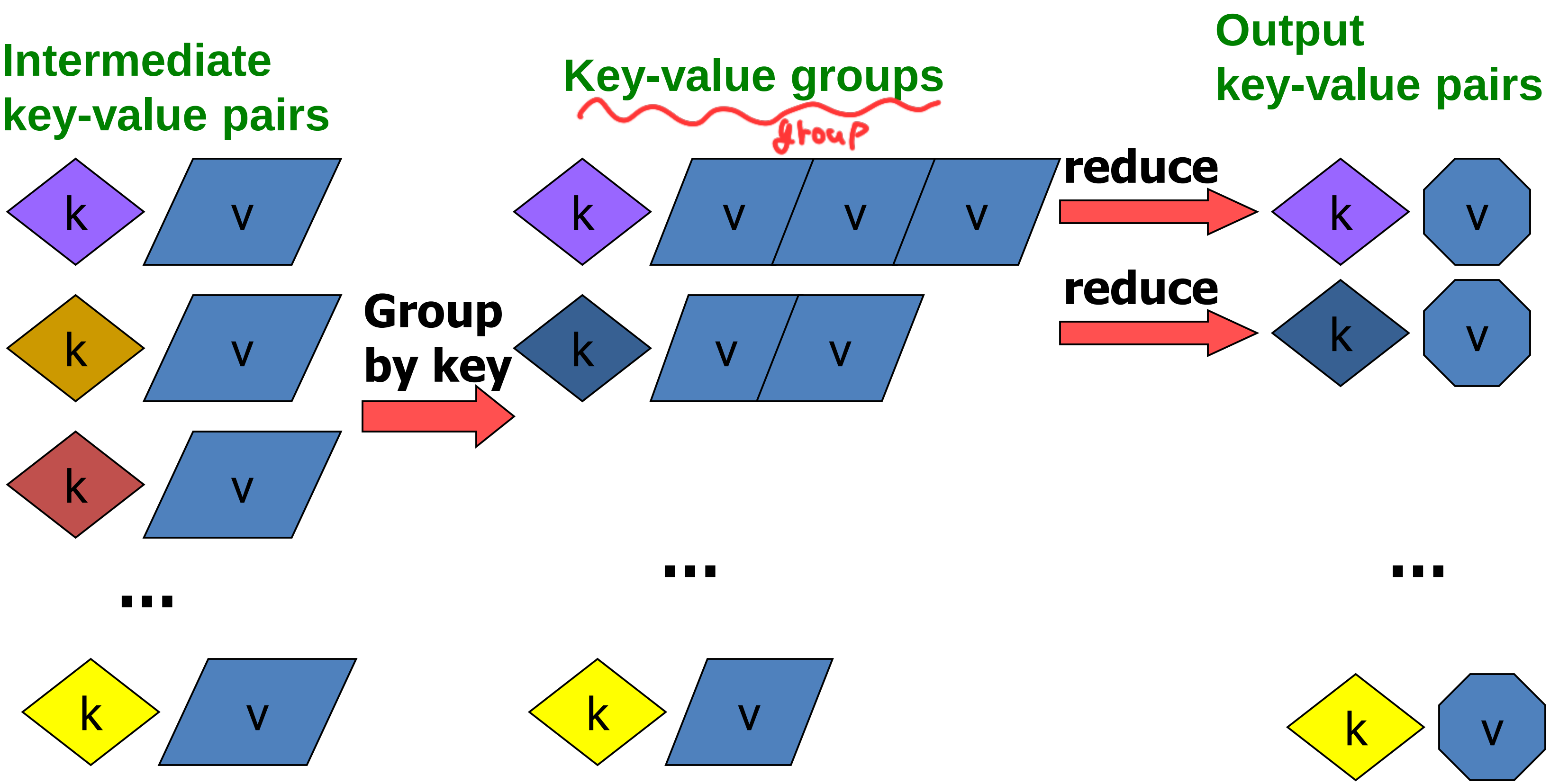
- **Input:** a set of key-value pairs

- Programmer specifies two methods:
  - **Map(k, v)** → <k’, v’>* set
    - 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’>*)** → <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: The Map Step



MapReduce: The Reduce Step



MapReduce: Word Counting

	Provided by the programmer		Provided by the programmer
	MAP: Read input and produces a set of key-value pairs	**Group by key:** Collect all pairs with same key	**Reduce:** Collect all values belonging to the key and output
The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. '"The work we're doing now -- the robotics we're doing -- is what we're going to need … ………………….	(The, 1) (crew, 1) (of, 1) (the, 1) (space, 1) (shuttle, 1) (Endeavor, 1) (recently, 1) ….	(crew, 1) (crew, 1) (space, 1) (the, 1) (the, 1) (the, 1) (shuttle, 1) (recently, 1) …	(crew, 2) (space, 1) (the, 3) (shuttle, 1) (recently, 1) …
Big document	**(key, value)**	**(key, value)**	**(key, value)**

Only sequential reads

Word Count Using MapReduce

```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

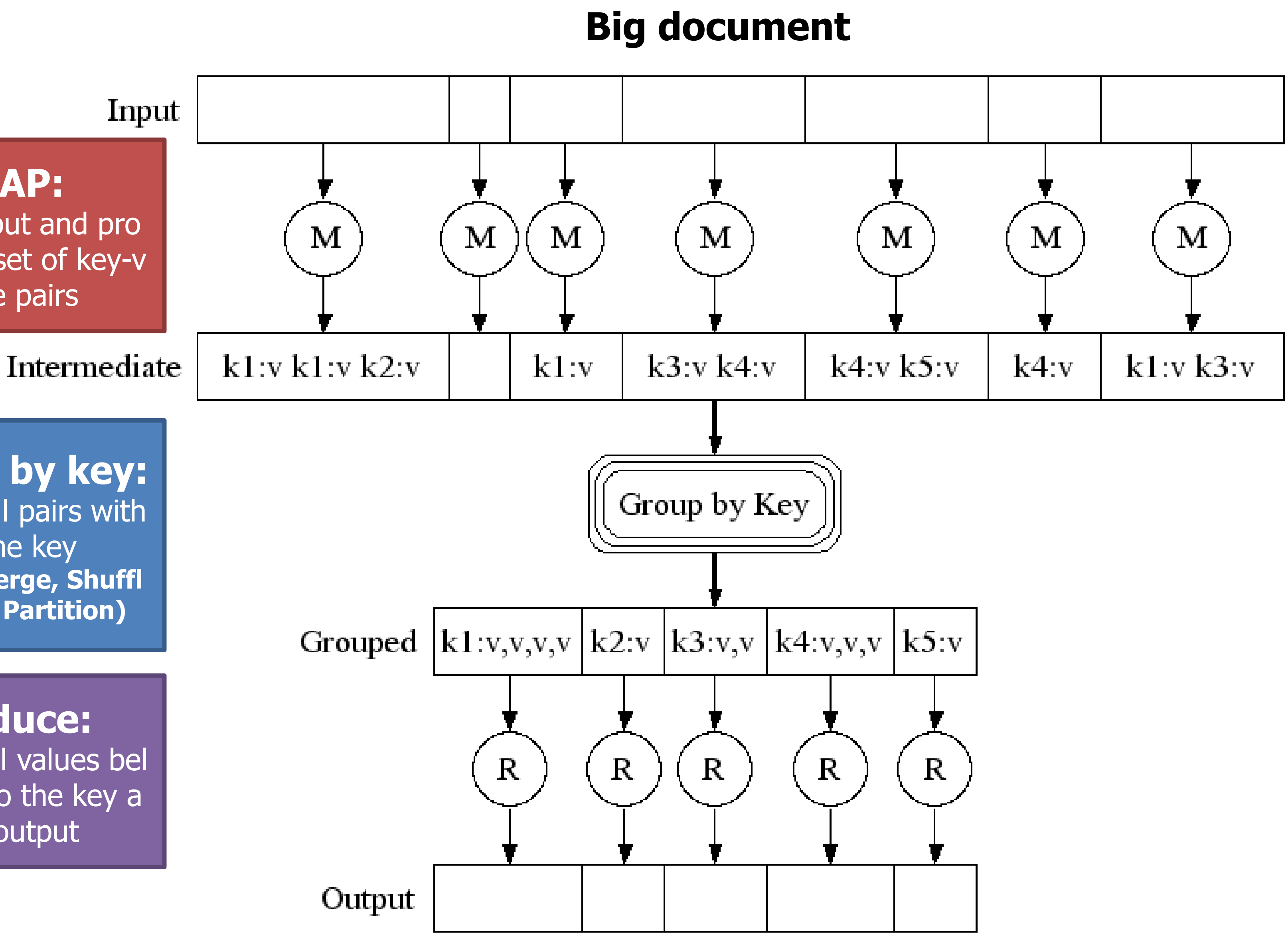
```
reduce(key, values):
// key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

MapReduce: A Logical 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

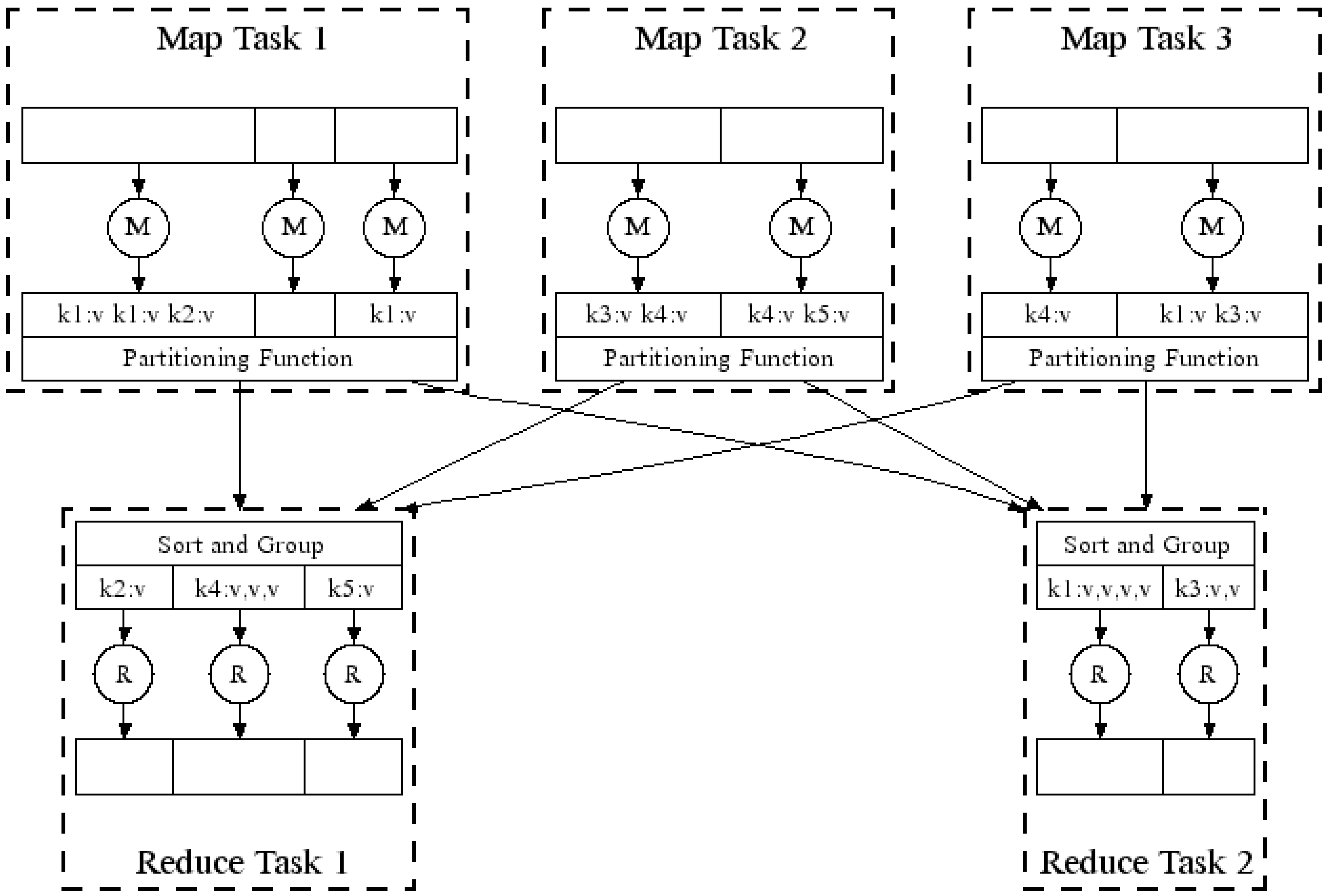


MapReduce: Environment

MapReduce 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

MapReduce: A Physical Diagram



All phases are distributed with many tasks doing the work

Problems Suited for MapReduce

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 MapReduce

- **Compute the natural join** ***R(A,B)*** $\bowtie$ ***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

$\bowtie$

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

Example: Join By MapReduce

- **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))* hash 값에 기반
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)* → key를 만들어 가는 과정
  - 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
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)

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

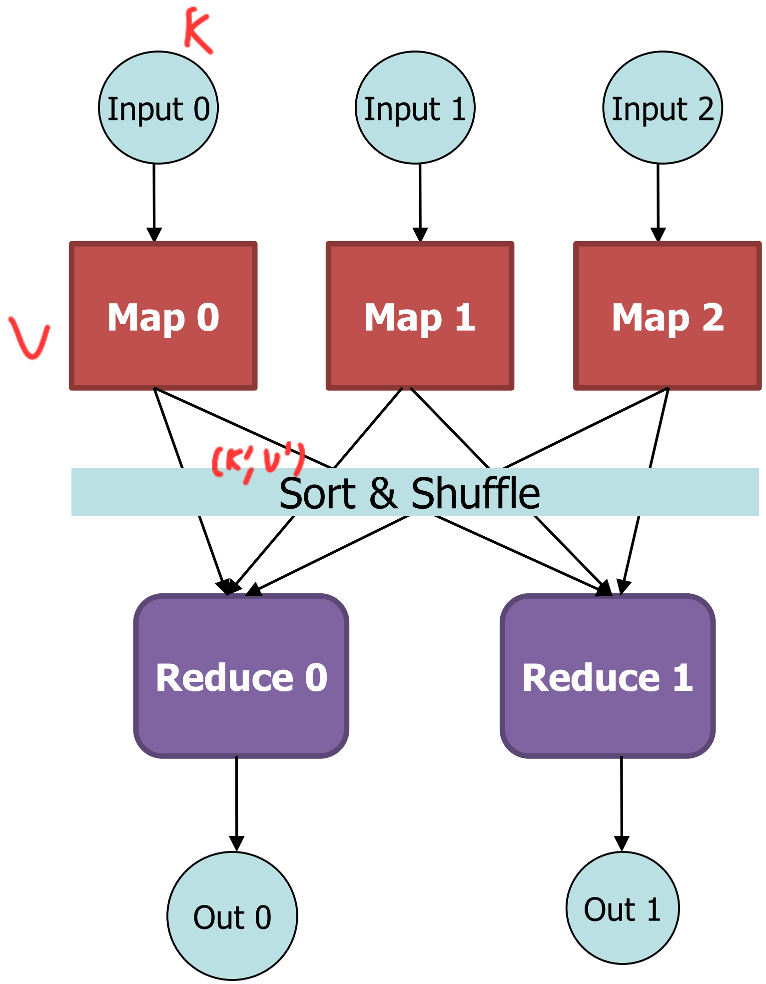
Design MapReduce Framework

Using MapReduce Framework

- 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



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

  File system

- Output is often input to another MapReduce task

  중간 결과 파일로 저장

CF) How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks

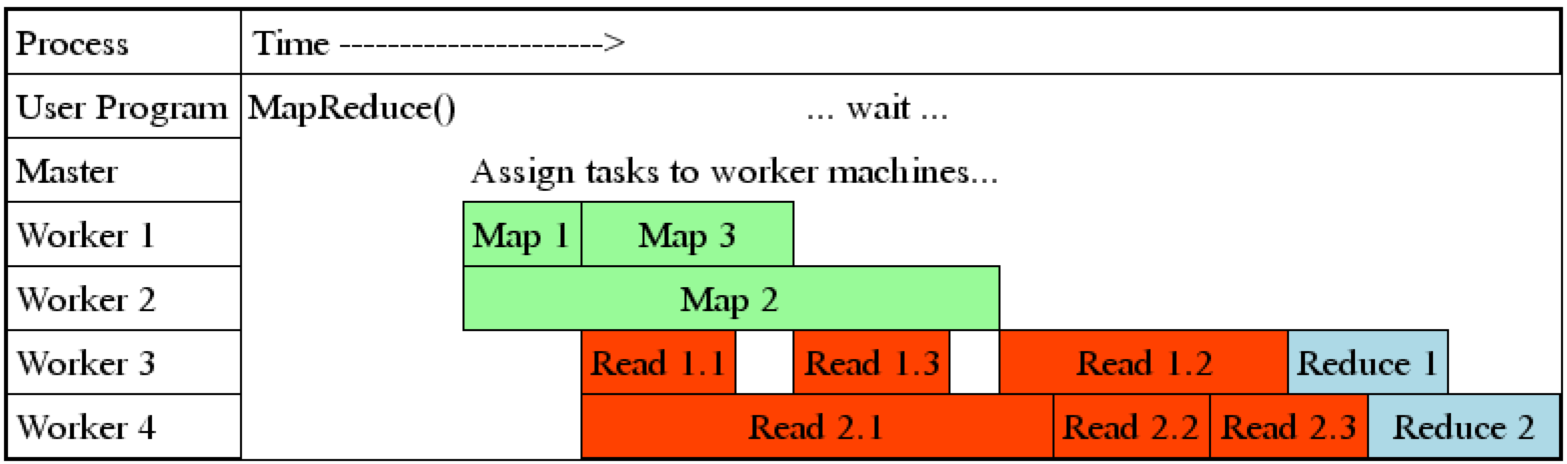
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures cost 계산 해야됨

- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

Task Granularity & Pipelining

task는 잘게 쪼갤수록 좋다

- **Fine granularity tasks:** map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing



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

map worker
reduce node
master node

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

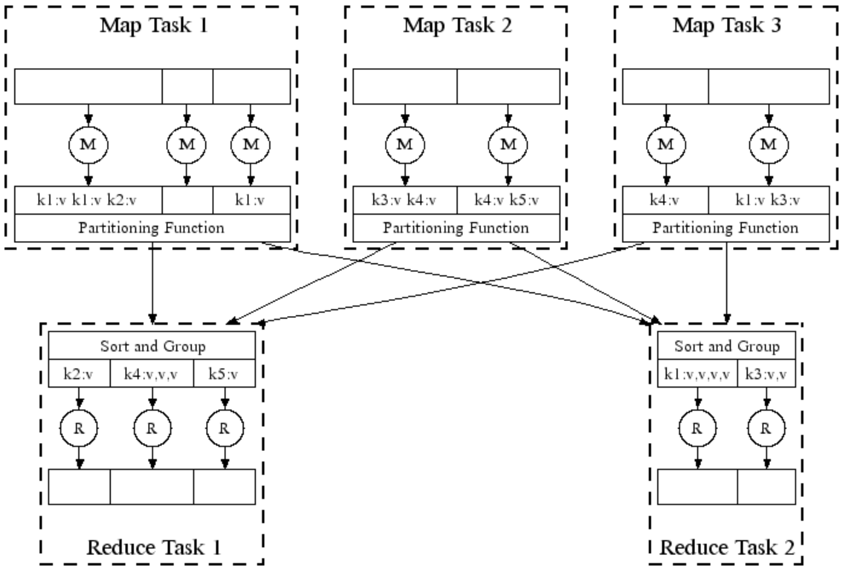
- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first "wins"
- **Effect**
  - Dramatically shortens job completion time

Refinement: Combiners

- Often a Map task will produce many pairs of the form *(k,v$_1$), (k,v$_2$), …* for the same key *k*
  - E.g., popular words in the word count example

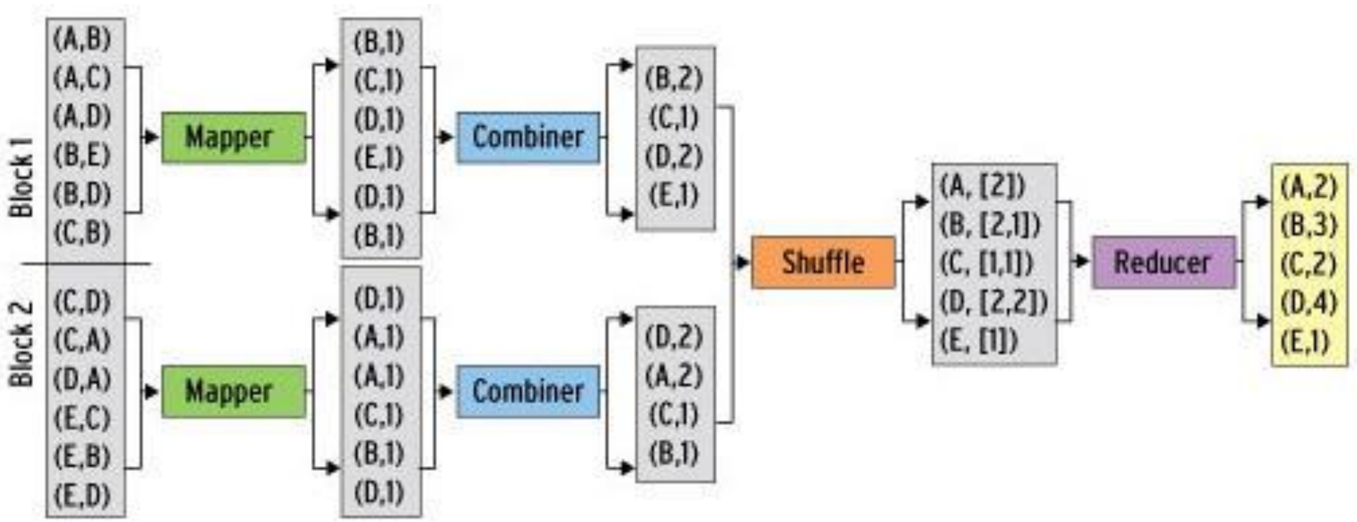
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function

- 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

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

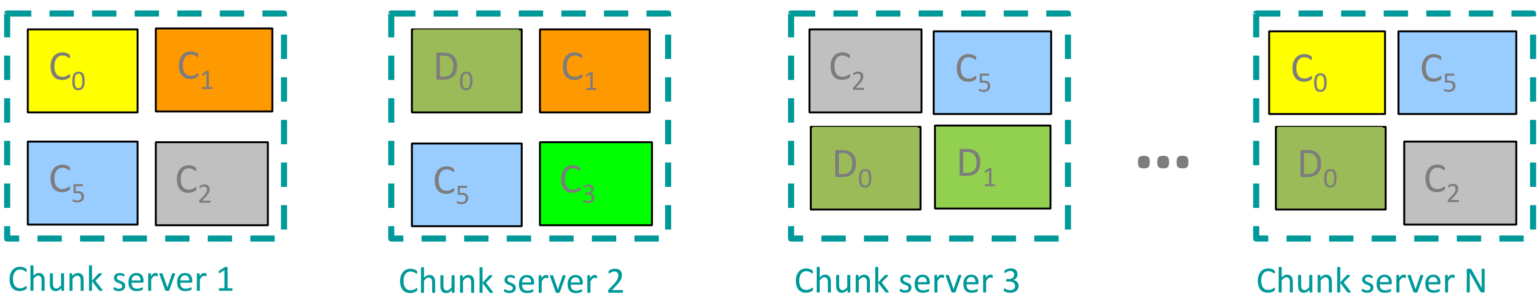
- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks

- **Master node**
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might 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 복제!!
  - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

Pointers and Further Reading

Pointers and Futher Reading

- 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 Sys tem (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 (cont'd)

- 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/
- 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])

Thank you!

Instructor: Daejin Choi (djchoi@inu.ac.kr)